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

D1.1– Report on targeted applications, their specifications, requirements

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:	IBM Research - Zurich (IBM)
Report Issue Date:	15/10/2018

Document History *(Revisions – Amendments)*	
Version and date	**Changes**
1.0 15/10/2018	Final Version
2.0 15/07/2019	Updated version for public release

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)	

European Commission

The MNEMOSENE project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement No 780215

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

Table of Contents

1. Introduction

1.1 Motivation

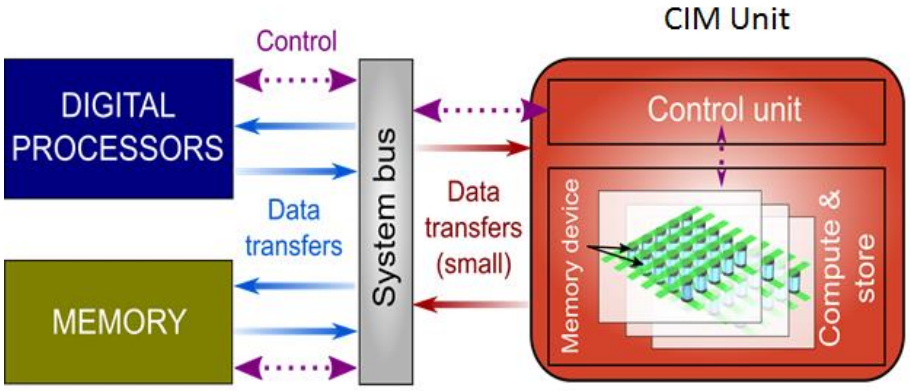

Figure 1: Computation-in-memory architecture is a new computing paradigm where certain computational tasks are performed in place in the memory itself by exploiting the physical attributes of the memory devices

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 (see Figure 1) 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[1,2].

1.2 Constituent elements

In a CIM architecture, unlike conventional memory, the memory device is an active participant in the computational memory. So it is natural to wonder what these devices look like. For decades, information was stored in the charge state of a capacitor. In the case of a DRAM, we have a capacitor in series with a transistor and in the case of flash memory, we have a capacitor coupled to the gate of a transistor. There are some recent attempts at using these conventional memory devices for computational memory[3]. However, it is widely believed that another class of memory devices are better suited for in-memory computing. In this class of

[1] Hamdioui, S., Xie, L., Nguyen, H.A.D., Taouil, M., Bertels, K., Corporaal, H., Jiao, H., Catthoor, F., Wouters, D., Eike, L. and van Lunteren, J., 2015, March. Memristor based computation-in-memory architecture for data-intensive applications. In *Proceedings of the 2015 design, automation & test in Europe conference & exhibition* (pp. 1718-1725). EDA Consortium.

[2] Sebastian, A., Tuma, T., Papandreou, N., Le Gallo, M., Kull, L., Parnell, T. and Eleftheriou, E., 2017. Temporal correlation detection using computational phase-change memory. *Nature Communications*, *8*(1), p.1115.

[3] Biswas, A. and Chandrakasan, A.P., 2018, February. Conv-RAM: An energy-efficient SRAM with embedded convolution computation for low-power CNN-based machine learning applications. In Proc. *IEEE International Solid-State Circuits Conference-(ISSCC), 2018* (pp. 488-490).

devices, information is stored in terms of the resistance state of the devices. The resistance state can be altered by the application of suitable electrical pulses. Hence they are referred to as resistive memory devices or memristive devices[4]. They are based on a range of physical mechanisms such as ionic drift, phase transition and magnetoresistance[5]. Depending on the physical attributes of these memristive devices various computational primitives could be implemented. For example, devices which can achieve a high or low resistance state can be used for logical operations. Devices that achieve a continuum of resistance values can be used to perform certain arithmetic operation using the Ohm's law and Kirchhoff's circuit laws. It is also possible to compute using the dynamic evolution of resistance values in these devices[2].

1.3 Computational primitives

In this report, we will focus our attention on two classes of computational primitives, logical operations and arithmetic operations.

In conventional CMOS, voltage serves as the single logic state variable. The input signal is a voltage signal. It is processed as a voltage signal and is output as a voltage signal. One approach to computational memory is to design logic using resistive memory devices [6]. A high resistance indicates logic "0" and low resistance indicates logic "1". This could enable the seamless integration of processing and storage. This introduces an additional state variable namely resistance. Various logical operations can be enabled based on the interaction between the voltage and resistance state variables. One particularly interesting characteristic of certain memristive logic families is statefulness. In this case the Boolean variable is represented only in terms of the resistance states. This concept can be extended to large cross-bar arrays where we can perform several such bit-wise operations in parallel. While stateful logic is very appealing due to the true "in-memory" nature of computation, the limited endurance of memristive devices pose a key challenge. Hence in this report, we will focus our attention of a non-stateful logic family that shows great promise.

The arithmetic operation that this report focuses on is is matrix-vector multiplication[7]. In order to perform $Ax = b$, the elements of A should be mapped linearly to the conductance values of resistive memory devices organized in a cross-bar configuration. The x values are encoded into the amplitudes or durations of read voltages applied along the rows. The positive and negative elements of A could be coded on separate devices together with a subtraction circuit, or negative vector elements could be applied as negative voltages. The resulting currents along the columns will be proportional to the result b. If inputs are encoded into durations, the result b is the total charge (e. g. current integrated over time). The property of the device that is used is the multi-level storage capability as well as the Kirchhoff circuit laws: Ohm's law and Kirchhoff's current law. The same cross-bar configuration can be used to perform a matrix-

[4] Chua, L., 2011. Resistance switching memories are memristors. *Applied Physics A*, *102*(4), pp.765-783.
[5] Waser, R. and Aono, M., 2007. Nanoionics-based resistive switching memories. *Nature materials*, *6*(11), p.833.
[6] Borghetti, J., Snider, G.S., Kuekes, P.J., Yang, J.J., Stewart, D.R. and Williams, R.S., 2010. 'Memristive'switches enable 'stateful'logic operations via material implication. *Nature*, *464*(7290), p.873.
[7] Burr, G.W., Shelby, R.M., Sebastian, A., Kim, S., Kim, S., Sidler, S., Virwani, K., Ishii, M., Narayanan, P., Fumarola, A. and Sanches, L.L., 2017. Neuromorphic computing using non-volatile memory. *Advances in Physics: X*, *2*(1), pp.89-124.

vector multiplication with the transpose of *A*. In this report we will present two applications using this computational primitive in the space of compressed sensing and deep learning.

1.4 Taxonomy of CIM architectures

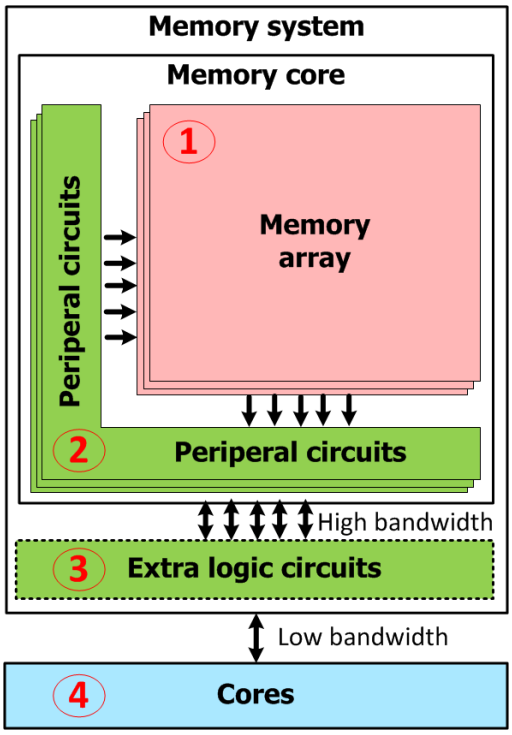

Figure 1. A typical CIM architecture and taxonomy based on where computation is performed

Figure 1 depicts a typical CIM architecture. The peripheral circuits feed input data to the memory array which transforms the input data and produces output results based on the configuration of the memory devices. The peripheral circuits receive these results and forward it to the extra logic circuits that can further process these results to produce final output from the CIM memory system. The programmable cores (or CPUs) can process the input or output data that cannot be easily processed in CIM system. Based on where the results are produced, the following terminology is used to indicate the actual computation position:

(1) Computation-In-memory – array (**CIM-A**)

(2) Computation-In-memory – peripheral (**CIM-P**)

(3) Computation-Out-memory – near (**COM-N**)

(4) Computation-Out-memory – far (**COM-F**)

In this report we will focus on CIM-A and CIM-P architectures.

1.5 Applications

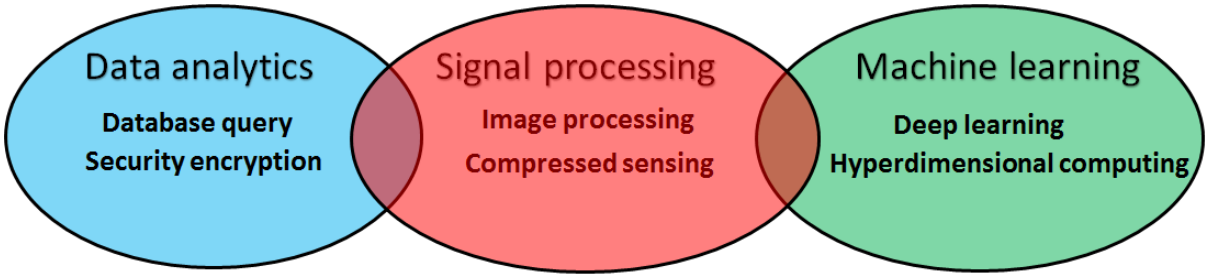

Figure 3: An overview of the applications that could benefit immensely from a CIM architecture

In subsequent sections, we will present numerous applications from the domains of data analytics, signal processing and machine learning/artificial intelligence that could benefit from a CIM architecture. An overview of the various applications is depicted in Figure 3.

2. CIM for Database application (query SELECT) using bitwise operations

In this section, we present applications based on logical operations that are performed in a CIM architecture. For each application, we first introduce the corresponding algorithm, the architecture, the primitive operation, and the working principle. The two computational primitives are bitwise-operation and dot-product operation.

2.1 Problem description

One of potential applications are big-data applications which includes high percentage of logical operations that perform poorly on conventional architectures due to high cache miss rates. In fact, some previous work has proposed quite a lot of applications with these characteristics such as database processing, graph processing, security encryption, and bio-sequencing[8,9]. Some potential examples are *QUERY SELECT* kernel (database applications) and *XOR encryption* kernel (security encryption).

- The *QUERY SELECT* database kernel performs the query-06 of the TPC-H benchmark[10], which includes 22 queries written in SQL language. The query-06 performs compare instructions to check if the requested data is available in the database or not. Note that a complete query execution may require other operations that must be performed on the host processor side. Currently, we only focus on the kernel that performs bitwise operations.

[8] Ahn, J., et al., A scalable processing-in-memory accelerator for parallel graph processing. ACM SIGARCH Computer Architecture News, 2016. 43(3): p. 105-117.
[9] Seshadri, V., et al. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. in Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.
[10] Council, T.P.P., *TPC-H, a decision support benchmark*. 2015.

- The *XOR encryption* kernel performs an XOR operation of a string sequence and a predefined (secret) key. It is used for one-time-pad cryptography[11]. In case the string sequence size is larger than the data-width of a single XOR instruction, multiple XOR instructions are executed using a loop with multiple iterations.

	Dist.	Size	Year
A	55	Large	2016
B	23	Medium	2014
C	43	Small	2015
D	60	Medium	2016
E	25	Medium	2000
F	34	Medium	2001
G	18	Small	2012
H	30	Small	2011

Figure 4 (a) Original dataset representing the information regarding some newly discovered stars

	A	B	C	D	E	F	G	H
Far	1	0	1	1	0	0	0	0
Near	0	1	0	0	1	1	1	1
Large	1	0	0	0	0	0	0	0
Medium	0	1	0	1	1	1	0	0
Small	0	0	1	0	0	0	1	1
New	1	0	0	1	0	0	0	0
Old	0	1	1	0	1	1	1	1
OR	1	0	1	1	0	0	0	0
AND	0	0	0	1	0	0	0	0

(b) Bitmap transposed representation of the dataset presented in (a)

In this report, we present an example of bitmap index schemes frequently used in QUERY SELECT database manipulations[12]. A bitmap index uses bitmaps (i.e., a vector of zeros and ones) to represent a database; generally they work well for low-cardinality columns, i.e., they have a limited number of distinct values. Figure 4(a) shows an example dataset with 8 entries, representing information of newly discovered stars. Each entry has three characteristics, i.e., distance (dist.), size and the year in which the star was discovered. Figure 4(b) presents the bitmap transposed representation of the same dataset, where the three characteristics (also called bins) are encoded into seven rows of zeros and ones; each column (i.e., A, B, C, D, etc.) is an entry while each row is a characteristic or bin. For example, a star with distance larger than 40 is defined as far, and otherwise as near. Similarly, a star that is discovered in 2016 is defined as new, and otherwise as old. Star A has a far distance, large size and is recently discovery; its bitmap is shown in the first column in Figure 4(b). The other stars are mapped in a similar manner. Typical database queries consist of searching for specific data patterns. These queries are carried out by performing bitwise operations (OR, AND, XOR) on the bitmaps.

[11] Yang, J., L. Gao, and Y. Zhang, *Improving memory encryption performance in secure processors.* IEEE Transactions on Computers, 2005. **54**(5): p. 630-640.

[12] Wu, K., Koegler, W., Chen, J., & Shoshani, A. (2003, July). Using bitmap index for interactive exploration of large datasets. In *Scientific and Statistical Database Management, 2003. 15th international conference on* (pp. 65-74).

2.2 Implementation with CIM architecture

2.2.1 Considered architecture

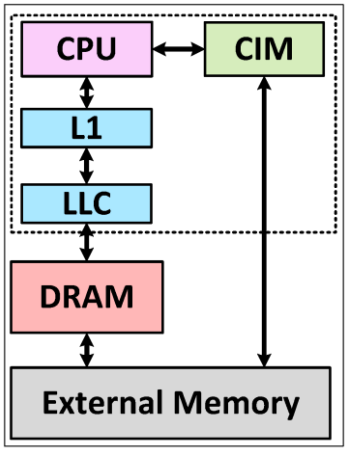

Figure 5: Considered architecture CIMA

The CIM accelerator (CIMA) architecture of Figure 4 consists of a conventional processor with cache, main DRAM memory, novel data-centric CIM core and an external memory. CIMA consists of a large non-volatile memory equipped with scouting logic. Both main memory and CIM core can fetch data from the external memory. CIM core is addressable from the processor and uses an extended address space. We assume that the data that is stored in the CIM core is not duplicated on the main DRAM memory; hence, no memory coherency schemes are required. The CIM core is initialized with data from the external memory, e.g., database(s); this initialization needs to be performed only once. Note that infrequent modifications are required when read-dominant applications are executed on CIM core.

In this report, we consider CIMA where CIM is used as an on-chip data-centric accelerator (as shown in Figure 5). CIM executes parts of the instructions that cannot be efficiently handled by conventional processors; for example, a loop that consists of simple operations on a huge data set. As accelerators are often read-favored, the impact on endurance is expected to be minimal. The data-centric accelerator differs from a traditional accelerator such as an FPGA or GPU in the amount of data that can be stored in the accelerator; this huge data storage alleviates the latency and energy cost required to move data back and forth from memory in traditional accelerators. However, it has also commonalities with the traditional accelerators. First, the performance gain is proportionately dependent on the number of parallel operations that can be performed simultaneously. Second, the memory coherence management is managed in a similar way, i.e., the accelerator can be seen from the processor as an extended memory space, similarly as for FPGA accelerators. Third, the accelerators can be executed in parallel with the host processor in case no data dependencies occur.

Bit-wise operation includes OR, AND, XOR gates that is performed within non-volatile memories by sensing (and *scouting*) the data stored in the memory, so-called *scouting logic*[13]. This reduces the number of write accesses into the non-volatile memory, hence, improves the device lifetimes, performance and energy saving in comparison with other memristive logic schemes. With scouting logic, it is feasible to implement a non-volatile memory that supports

[13] Xie, L., et al. Scouting Logic: A Novel Memristor-Based Logic Design for Resistive Computing. in IEEE Computer Society Annual Symposium on VLSI.

Computation-in-Memory (CIM). In other words, the memristor-based memory module is equipped with one or more scouting logic units.

2.2.2 Scouting logic

Scouting logic executes logic operations by modifying the read circuitry. 5(a) shows a resistive memory with two cells based on 1T1R cells. Normally when a cell is read, (e.g., memristor M1), a read voltage V_r is applied to its row and switch S_1 is activated. Subsequently, a current I_{in} will flow through the bit line to the input of the sense amplifier (SA). This current is compared to the reference current I_{ref}. If I_{in} is greater than I_{ref} (i.e., when R_1 has a low resistance R_L), the output of the SA changes to V_{dd} (logic 1). Similarly, $I_{in}<I_{ref}$ (i.e., when R_1 has a high resistance R_H), the output changes to logic 0. For proper operations, I_{ref} should be fixed between the high and low current as depicted in the top left part of Figure 5(b).

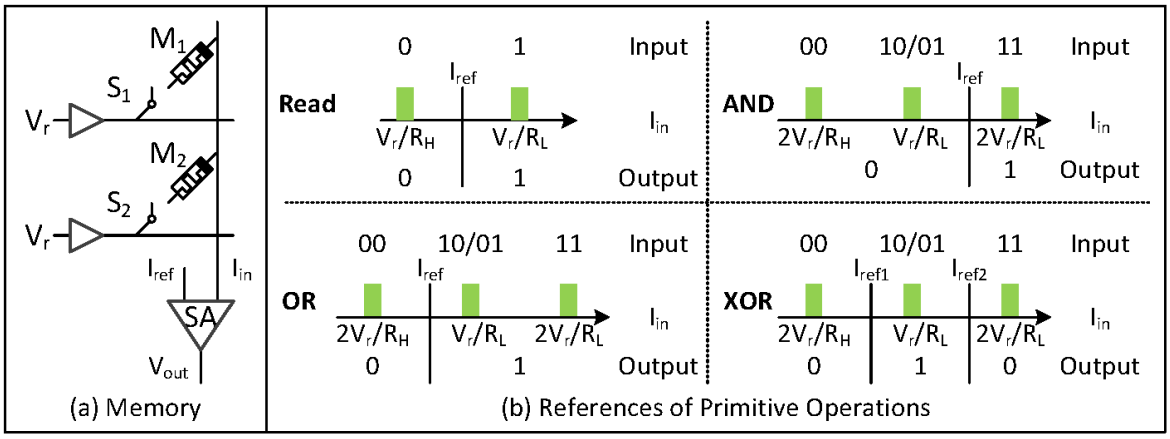

Figure 6: Scouting logic. The input logical state variables are represented in terms of the resistance values of the memristive devices. The devices are read simultaneously. The output logical state variable is a voltage signal generated by a comparator circuit that compares the current flowing along the column with a reference current. Different logical operations can be implemented by choosing the appropriate reference current.

Inspired by this read operation, we demonstrate how to implement OR, AND and XOR scouting logic gates, which are frequently used bitwise logic operations. Instead of reading a single memristor at a time, scouting logic activates two (or more) inputs of the gate simultaneously (e.g., M_1 and M_2 in Figure 6(a)). As a result, the input current to the sense amplifier is determined by the equivalent input resistance ($R_1//R_2$). This resistance results in three possible values: $R_L/2$, $R_H/2$ and $(R_L//R_H)\approx R_L$. Hence, the input current I_{in} also can have only three values. By changing the value of I_{ref}, different gates can be realized. To implement an OR gate, I_{ref} should be set between $2V_r/R_H$ and V_r/R_L as depicted in the left bottom part of Figure 6(b)). As a result, the output is 0 only when $R_1//R_2=R_H/2$. Similarly, to implement an AND operation, I_{ref} should be set between $2V_r/R_L$ and V_r/R_L. The XOR operation needs two references and the output is logic 1 only when $R_1//R_2\approx R_L$.

2.2.3 Working principle

The bitmaps (as shown in Figure 4(b)) can be efficiently mapped to a non-volatile memory that uses scouting logic to perform queries. Typical queries are `*find a star that satisfies {far or large}*' and `*find a star that satisfies {far, medium and new}*'. A bitwise *OR* operation of the row *far* and *large* produces the result of the first query which are the stars *A, C, D*. The result of the second query can be obtained by the bitwise *AND* operations of rows *far*, *medium*, and

new, the result is shown in the row *AND* at the bottom of the Figure 4(b) where star *D* satisfies this query.

2.3 Comparative study

In order to evaluate CIMA, we would like to compare its performance, energy consumption and area with a conventional architecture. To be able to realize this, we create two models one for the conventional and one for CIMA architecture. The models can be used to evaluate different applications.

We characterize an application by the fraction of logical instructions n_l, other instructions n_r (arithmetic, conditional, etc.) and the fraction of their corresponding memory accesses m_l, m_r, respectively. Note that $n_l+n_r=1$.

For the **conventional architecture**, we use the Intel Xeon E5-2680 multicore as a baseline. We assume the following:

- The number of cores is $n_p=4$, each with a frequency of 2.5GHz.
- Each core contains (i) an ALU that is capable of executing non-memory instructions (logical, arithmetic, conditional instructions) in one cycle, (ii) a two level cache (L1 of 32KB and L2 of 256KB) with access latencies of one and two cycles[14], and a miss rate of mr_{L1} and mr_{L2}, respectively.
- The cores share a main DRAM memory of 4GB with an access latency of 175 cycles (165 cycles for communication and 10 cycles for retrieving the results). The page fault rate of the main DRAM memory is $pf_{dr}=0.007$ with a penalty of 800 cycles[15].

For the **CIMA architecture** (see Figure 5), we assume the following:

- A single host processor has the same characteristics as an individual core in the conventional architecture. It contains (i) an ALU that executes non-logical instructions in one cycle, (ii) 32KB L1 cache and 256KB L2 cache, (iii) 1GB DRAM, and (iv) a CIM core with $n_a=786,432$ parallel memory arrays (each with a size of 32 columns x 1024 rows), that occupies an area which is equivalent to the size of 3GB DRAM.
- Unidirectional communication between the processor and CIM core is assumed to be equal to L1 access latency (i.e., 1 cycle). Note that both L2 cache and CIM core are on-chip.
- A logical instruction takes 9.3ns on CIM core based on the VS; this is equivalent to 4 CPU cycles.

Based on the above assumptions, we evaluate the two architectures in terms of (normalized) delay and energy. We investigate the impact of L1 and L2 cache miss rates, the problem size and the percentage of logical instructions accelerated by CIM core on the performance of both architectures. Using this analytical evaluation, we can quickly perform a design space exploration.

Figure 7 shows the performance metric of the conventional architecture (red planes) with respect to CIMA architecture (green planes) for different problem sizes (PS) (either terabyte

[14] Esmaeilzadeh, H., et al. Dark silicon and the end of multicore scaling. in Computer Architecture (ISCA), 2011 38th Annual International Symposium on.

[15] Yamauchi, T., L. Hammond, and K. Olukotun. *A single chip multiprocessor integrated with DRAM*. in *Workshop on Mixing Logic and DRAM, held at the 24th International Symposium on Computer Architecture*

(TB) or petabyte (PB)) and percentage of accelerated instructions (%Acc.) (ranging from 30% to 90%). For a given problem size, the performance speed-up of CIMA architecture increases for larger percentages of the instructions executed on CIM core. This can be clearly observed as the gap between the red and green planes increases.

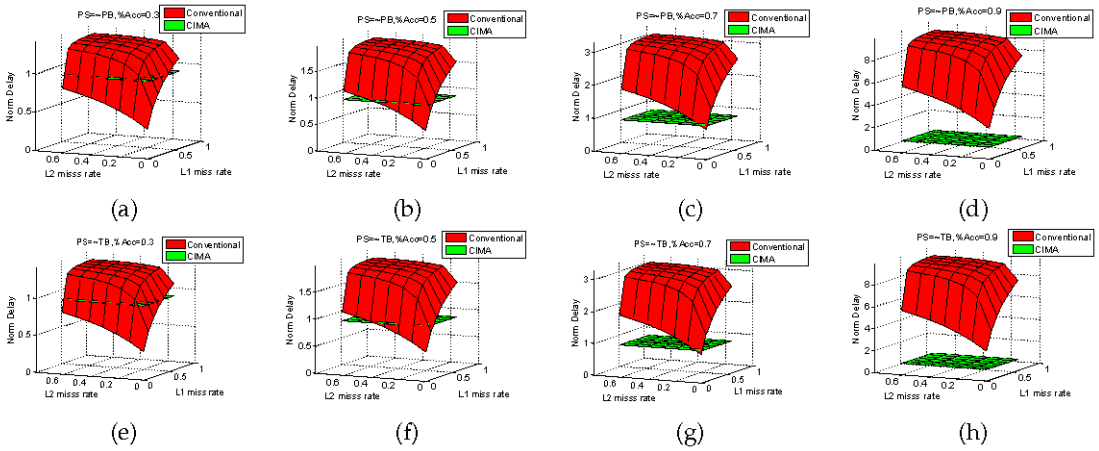

Figure 7: Analytical results of the performance (delay) metric for the conventional and CIMA architectures

CIMA architecture outperforms ideal applications for the conventional architecture (i.e., both L1 and L2 cache miss rates are 0) only when the percentage of accelerated instructions is higher than 75%. However, when the miss rates increases, a much lower percentage of accelerated instructions is required to reach the break-even point. For example, when the miss rates of L1 and L2 are around 10%, 50% of the instructions have to be accelerated to realize a higher performance on CIMA architecture. Note that for big-data applications, the miss rate is often around 6-15%[16]. Moreover, it has been shown that 30% of a database application could be accelerated using in-memory computing[17]. This corresponds to the case when the application perform badly on conventional architecture with high cache miss rate (as shown in Figure 7 (a)). Note that high cache miss rates result in a longer memory access latency in the conventional architecture. CIMA architecture does not suffer much from this as computing takes place within the memory (i.e., CIM core).

[16] Sánchez, F., et al. Performance analysis of sequence alignment applications. in Workload Characterization, 2006 IEEE International Symposium on.

[17] Li, S., et al. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. in Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE.

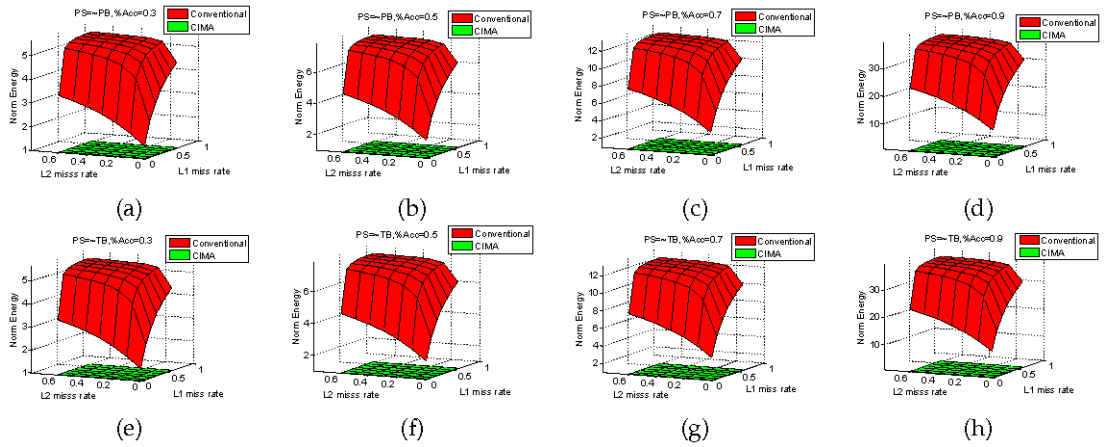

Figure 8: Analytical results of the energy metric for the conventional and CIMA architectures

Similarly, Figure 8 shows the energy metrics for both architectures. Overall, similar trends are observed with respect to the percentage of accelerated instructions and problem size. However, the energy consumption of CIMA architecture is always lower, irrespective of the cache miss rates. In case 30% of the instructions are accelerated on CIM core, the conventional architecture consumes nearly twice as much energy for the same problem size. This grows up to 10x to 40x in case 90% of the instructions are accelerated on CIM core. The high energy consumption of the conventional architecture can be partly attributed to the data movement back and forth between the CPU and the memory hierarchy. In addition, both cache and DRAM suffer from much higher leakage current as compared with non-volatile technology based CIM core. To the best of our knowledge, CIMA architecture is the first architecture that integrates a conventional architecture with a non-volatile memory based core. In comparison to the state-of-the-art, the proposed architecture realizes significant improvements, despite the low performance of individual memristors. The improvement could be much higher if more instructions can be accelerated, and/or if the CIM core performance can be improved. For example, in case 90% of the instructions can be accelerated on CIM core, one order of magnitude improvement for the energy-delay product (combined of energy and delay metric) can be realized. Furthermore, applications with a bad data locality and high data volume can be significantly benefit from the proposed architecture.

3. CIM for Image Processing

The next generation of image and video processing kernels often exhibit a mix of regular and irregular (or data-dependant) memory accesses. Moreover, they require data access which goes beyond the immediate local neighbours. Typically, they need a medium-size neighbourhood around the current pixel access. Typical values can be from 7x7 up to 11x11 pixels of 23 bits (in the case of colours); and these do not directly fit in the local register-files, so they need to be accessed from SRAM caches or scratchpad memories. This limits the efficient mapping of these kernels on modern GPUs. In this section we present the 'Guided Image Filtering' application kernels for their efficient execution on CIM architectures. In this application, the CIM architecture is of CIM-P type as per the taxonomy presented earlier.

3.1 Guided Image Filtering

The guided image filtering application comprises a guidance image 'I', a filtering input image 'p', and an output image 'q'. Both the guidance image I and the input image p act as input to

the application, and as a special case, they can even be identical. Figure 9 illustrates the bilateral and guided filtering process.

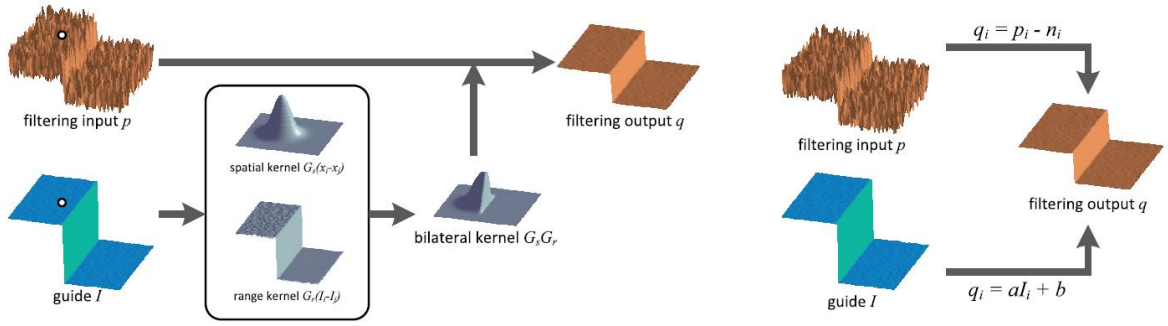

Figure 9: Bilateral Filtering Process and the Guided Filtering Process

In summary, the guided image filtering application:

- Computes the filtering image output based on a guidance image
- Edge-preserving smoothing operator with improved performance near edges
- Enables new filtering applications like dehazing and guided feathering
- Has a mix of vector and irregular memory accesses

He et al. presents a detailed description of the guided image filtering application[18]. This document reviews few image processing kernels from the guided image processing application that are proposed for the next generation high resolution TV and video applications. In the near future the TV raster/image sizes are expected to go to 8K UHD (7680×4320 pixels) from the current 4K (3840 x 2160 pixels or 4096 x 2160 pixels) with typical video capture frame rates of 30fps or 60fps. For these TV applications the image processing kernel sizes will also change from the current 3x3 and 5x5 to 7x7 and 11x11.

3.2 Image processing kernels amenable to CIM

This section shows a few selected image processing kernels and the segments from these kernels that are more efficient for implementation on CIM architectures. These kernels have been implemented in Quasar programming language[19]. The Quasar language, which is developed at the Image Processing and Interpretation Research Group, Ghent University (http://ipi.ugent.be), compiles and executes the kernel on GPUs. Future plans include collaborating with this Image Processing and Interpretation Research Group which is also part of the IMEC holding (MNEMOSENE project partner).

As shown in the code snippets below (Figures 10-11), each of these kernels contains segments that are better suited on CIM architectures compared to the modern GPU platforms. Hence, this is a very promising application domain to further focus on with the MNEMOSENE architecture work packages.

[18] K. He, J. Sun and X. Tang, "Guided Image Filtering," in IEEE Transactions on Pattern Analysis & Machine Intelligence, vol. 35, no. , pp. 1397-1409, 2013. doi:10.1109/TPAMI.2012.213
[19] https://quasar.ugent.be/bgoossen/quasar

```
function [] = cross_bilateral_filter(rgb_image : cube, depth_image : mat, depth_image_out : mat, r : int, h :
scalar)
  [M,N] = size(depth_image_out, 0..1)
  for py=0..M-1
    for px=0..N-1
      new_depth = 0.0
      total_weight = 0.0
      rgb_central = rgb_image[py, px, 0..2]

      for dy=-r..r-1
        for dx=-r..r-1
          diff = sum(abs(int(rgb_central - rgb_image[py + dy,px + dx, 0..2])))
          weight = exp(-(h/r^2) * diff)
          new_depth += weight * depth_image[py + dy, px + dx]
          total_weight += weight
        endfor
      endfor
      depth_image_out[py, px] = new_depth / max(total_weight, 1.0)
    endfor
  endfor

endfunction
```

Better suitable than GPU for implementation on CiM architectures

Figure 10: Cross Bilateral Filter Kernel

```
function [] = diagonal_fir_filter(im : cube)
  {!function_transform enable="imperfectloop"}
  [M,N] = size(im,0..1)
  for m=0..M-1
    for n=0..N-1
      for p=0..2
        im[m,n,p] = 0.3 * im[m-1,n-1,p] + 0.4 * im[m-2,n-2,p] + 0.3*im[m,n,p]
      endfor
    endfor
  endfor
endfunction
```

- Can be implemented either on GPU or CiM architectures
- CiM architectures with line buffers are more suitable for this

Address beyond current processing line requires multiple line buffers (6, 10, 14 or 22) depending on the kernel size

Figure 11 (a): Diagonal FIR Filter Kernel

```
function masks = create_anisotropic_gaussian_masks(sigma_x : scalar, sigma_y : scalar)
  n = 2
  sz = 5
    % Compute the filter kernel
  fc_x = exp(-0.5/sigma_x^2*(-n..n).^2)
  fc_x = fc_x / sum(fc_x)
  fc_y = exp(-0.5/sigma_y^2*(-n..n).^2)
  fc_y = fc_y / sum(fc_y)

  fc = transpose(fc_y)*fc_x
  masks = zeros(sz,sz,17)
  masks[:,:,0] = fc %first iteration: 0 degrees rotation
  %rotate 360/(2*16) degrees on every iteration
  for i = 1..15
    %avoid weird boundary effects -> pad with zeros
    fc_bound = zeros(sz+4,sz+4)
    fc_bound[2..sz+1,2..sz+1] = fc

    theta = -11.25*i*(3.14159265359/180.0)
    tf_m_bound = imrotate(fc_bound, theta, "keep", "zero", "bilinear")
    masks[:,:,i] = tf_m_bound[2..sz+1,2..sz+1]
  endfor

  %isotropic gaussian at masks[:,:,6]
  sigma = (sigma_x+ sigma_y)/2.

  fc_x = exp(-0.5/sigma^2*(-n..n).^2)
  fc_x = fc_x / sum(fc_x)
  fc_y = exp(-0.5/sigma^2*(-n..n).^2)
  fc_y = fc_y / sum(fc_y)

  fc = transpose(fc_x)*fc_y
  masks[:,:,16] = fc

endfunction
```

Can be implementation either on GPU or CiM architectures

Better suitable than GPU for implementation on CiM architectures

GPU and CiM suitable code in a single kernel loop makes it ideal for CiM implementation

Figure 11 (b): Creation of Anisotropic Gaussian Masks

3.3 Implementation with CIM architecture (confidential, patent submission ongoing)

Based on the characteristics of the complex modern multimedia (video, graphics, image etc) signal processing applications, as described above in subsection 3.1 and 3.2, we have a derived preliminary proposal for the corresponding CIM architecture requirements. We believe that an important target domain for CIM architectures can be the support of partly irregular index operations on the multi-dimensional arrays that are common in multimedia applications. In that domain, a crucial class of CIM operations to enable this is associated with the addressing of the data that would be stored in the CIM memory array. In particular, current memory decoders will not support these in a streaming energy-efficient fashion. Instead, loops with complex conditions and much instruction overhead would have to be allowed on CPUs and even more so on GPUs. Especially, the modern GPUs are not suited for any irregular indexing because they have been fully optimized for regular streaming signal processing in a wide vector operation/instruction set.

CIM implementation details are confidential and are not reported in the public version of the present document.

4. CIM for compressed sensing and recovery

In sections 4 and 5, we will focus on applications that employ the computational primitive of matrix-vector multiplication using a memristive cross-bar array. In this section, we will focus on the task of compressed sensing and recovery.

4.1 Problem description

Reconstruction of a sparse high-dimensional signal from low-dimensional noisy measurements, for example received by sensor arrays, is used in many application fields, including radio interferometry for astronomical investigations, and magnetic resonance imaging (MRI), ultrasound imaging, and positron emission tomography for medical applications. Such reconstruction can also be applied for devices performing audio restoration or imaging, such as a mobile phone camera sensor. In the latter, one can significantly reduce the acquisition energy per image, or equivalently increase the image frame rate, by capturing only few measurements (e.g. 10%) instead of the whole image, while still being able to recover the original image accurately.

Unfortunately, high-performance sparse signal recovery algorithms typically require a significant computational effort for the problem sizes occurring in most practical applications. While the computational complexity is not a major issue for applications where off-line processing on CPUs or graphics processing units (GPUs) can be afforded, it becomes extremely challenging for applications requiring real time processing at high throughput or for implementations on battery-powered (e.g., mobile) devices. Moreover, implementations that can deal with extremely large signals may be desirable in applications where signals are received by large sensor arrays, for example the Square Kilometer Array, where the signal size may be on the order of 10^8.

In practically all the applications mentioned above, the observation model can be formulated as

$$y = Ax_0 + w$$

where $A \in \mathbb{R}^{MxN}$ is a known measurement matrix, $x_0 \in \mathbb{R}^N$ is the signal of interest, $y \in \mathbb{R}^M$ is the measurement data vector and $w \in \mathbb{R}^M$ represents the measurement noise. The goal is to recover x_0 from y when $M < N$. A first order approximate message passing (AMP) technique for reconstructing x_0 given y (Donoho et al., PNAS,2009) may be represented as

$$z^t = y - Ax^t + \frac{N}{M} z^{t-1} \langle \eta'_{t-1}(A^* z^{t-1} + x^{t-1}) \rangle,$$

$$x^{t+1} = \eta_t(A^* z^t + x^t),$$

where $x^t \in \mathbb{R}^N$ is the current estimate of x_0 at iteration t, $z^t \in \mathbb{R}^M$ is the current residual, A^* is the transpose of A, $\eta_t(\cdot)$ is a function, $\eta'_t(\cdot)$ its derivative, $\langle \cdot \rangle$ denotes the mean and $x^0 = 0$. The final value of x^t provides the estimate of x_0. The AMP technique may be equivalently formulated as an iterative thresholding process, which may provide the reconstruction power of other approaches, when sparsity of the solution may be assumed, at a much lower complexity. The AMP algorithm has a relatively simple formulation and requires only multiplications and additions, making it suitable for a memristive non-von-Neumann implementation.

For example, the AMP technique may be used to reconstruct a random vector $x_0 \in \mathbb{R}^N$ that is sparse, i.e., only $K < N$ vector elements are nonzero. In this case, a nonlinear function $\eta_t(\cdot)$ may be preferably used, e.g.,

$$\eta_t(x) = \begin{cases} x - \tau_t, & x \geq \tau_t \\ x + \tau_t, & x \leq -\tau_t \\ 0, & \text{otherwise} \end{cases}$$

where $\tau_t^2 = \|z^t\|_2^2 / M$.

4.2 Implementation with CIM

The key idea of realizing compressed sensing using CIM relies on the encoding of the elements of A as conductance values of memristive devices organized in a crossbar array, as depicted in Figure 13. One possible method to program the conductance values is by an iterative program-and-verify procedure. The compressed measurements y are acquired by applying x_0 as voltages to the crossbar rows via digital-to-analog conversion, and obtaining y through analog-to-digital conversion of the resulting output currents at columns. The positive and negative elements of A can be coded on separate devices together with a subtraction circuit, whereas negative vector elements can be applied as negative voltages.

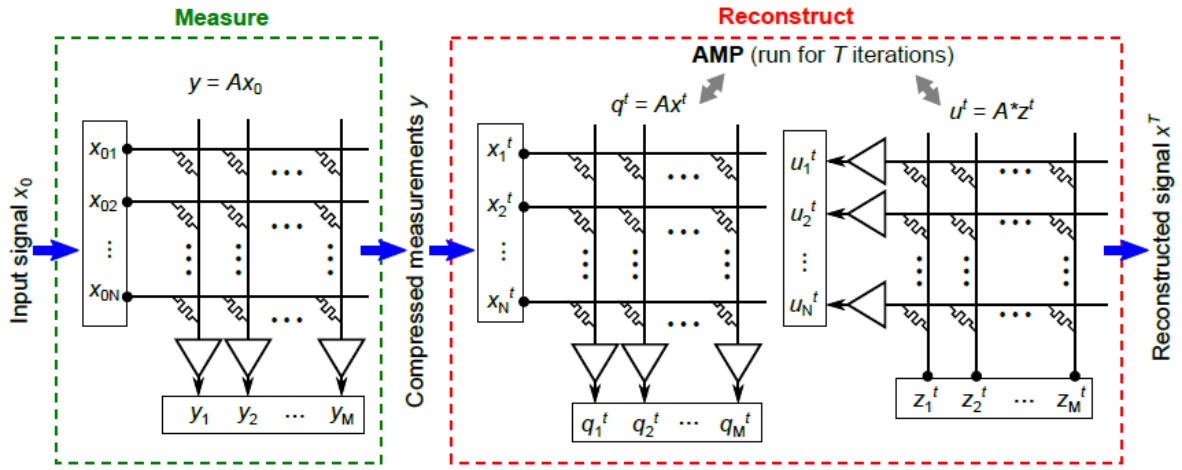

Figure 13: Proposed CIM implementation of compressed sensing with AMP recovery. A $\mathbf{N \times M}$ memristive crossbar encoding the measurement matrix A is used to acquire the compressed sensing measurements and to realize the matrix-vector computations $\mathbf{q^t = Ax^t}$ and $\mathbf{u^t = A^*z^t}$ of the AMP recovery algorithm. The matrix A is programmed only once in the crossbar and the same crossbar is used for the measurements and reconstruction.

Once the matrix A is programmed in the crossbar array and the measurements y are obtained, the AMP algorithm can be implemented as illustrated in Figure 14. The AMP algorithm is run in a dedicated processing unit, while the computation of $q^t = Ax^t$ and $u^t = A^*z^t$ is performed using the (same) crossbar array. The vector q^t is computed by applying x^t as voltages to the rows and reading back the resulting currents on the columns, and u^t by applying z^t as voltages to the columns and reading back the resulting currents on the rows.

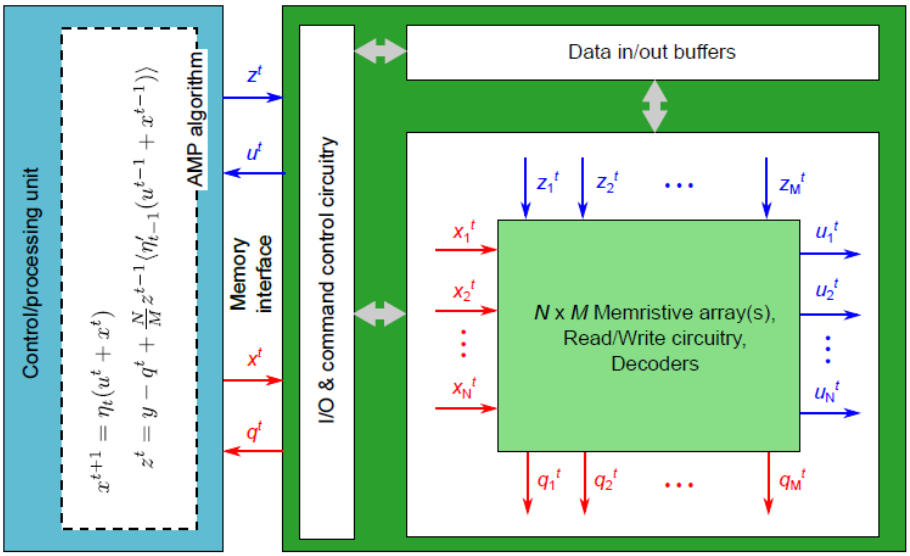

Figure 14: Architecture for CIM implementation of AMP. The AMP algorithm is implemented in a dedicated processing unit and the matrix-vector multiplications $q^t = Ax^t$ and $u^t = A^*z^t$ are performed using memristive array(s).

In the AMP algorithm, ignoring the $\eta_t(\cdot)$ and $\eta_t'(\cdot)$ functions, the main computational cost comes from the matrix-vector multiplications Ax^t and A^*z^t which both require O(MN) operations for dense A. The other operations in the AMP algorithm are vector additions and

multiplications which require O(N) operations. Thus, one could potentially reduce the complexity of AMP from O(MN) to O(N) by performing Ax^t and A^*z^t in memristive arrays, assuming that $\eta_t(\cdot)$ and $\eta_t'(\cdot)$ involve only O(N) or less operations. The expectation is that in a memristive crossbar, matrix-vector multiplications can be performed with constant time complexity O(γ), where γ is independent of the crossbar size. The reason is that the computation is performed in parallel through Kirchhoff's circuit law locally at the same place where the matrix data are stored. The precise value of γ will depend on the read current settling time and the time required to digitize the current by the peripheral circuitry. Consequently, larger crossbars may eventually lead to higher γ if some of the readout circuitry must be shared across columns/rows and multiplexed.

4.3 Energy analysis

To quantify the potential energy gains of the CIM implementation over a digital design, based on the figures currently achieved with our prototype PCM chip, we made an FPGA design that operates at the same speed and the same precision at which we expect a PCM-based crossbar to perform. In the AMP algorithm, the matrix-vector multiplications are the most expensive operations, so we compared the memristive crossbar analog multiplier with a 4-bit FPGA multiplier design. We focus in this analysis on the energy drawn by the computational units and disregard the time and power consumption of the data transfers.

The design of the FPGA architecture is based on a configurable number of dot-product units. As depicted in Figure 15, each unit consists of a BlockRAM module to store one or multiple rows of matrix *A* and a 5-stage processing pipeline to compute the dot-product. The matrix *A* is preloaded into BlockRAM modules during the initialization phase. In the computation phase, the right-hand side vectors get streamed to all dot-product units.

The data width of the incoming matrix and vector data is assumed to be 4-bit. As the BlockRAM modules of the FPGA device have a native data width of 32-bit, we compute the dot-product in a vectorized fashion. In the first stage, eight 4-bit numbers of the matrix fetched from local RAM are multiplied with a 8x4-bit chunk of the right-hand-side resulting in a vector of eight 8-bit signed integer values. The individual products are then reduced to a single 11-bit number in a 3-stage adder tree. In the final stage we accumulate the result in a 14-bit register.

Table 1 shows the resource utilization, frequency and estimated dynamic on-chip power consumption of a Xilinx Kintex Ultrascale FPGA device (xcku115-flva1517-2-e, built with TSMC's 20nm system on chip technology) for a 1024 dot-product units design. The 1024-unit design allocates the logic (look-up tables, LUT) and memory (BlockRAM, BRAM) resources to almost 50% leaving a reasonable amount of space on the device to implement the peripheral logic and buffers.

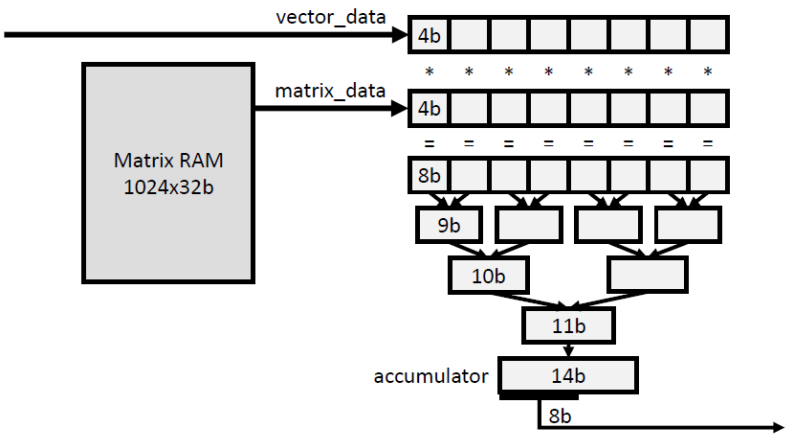

Figure 15: Dot-product unit architecture with matrix row memory and 4-bit input based processing pipeline.

The achieved clock frequency for this design on the xcku115 chip is 200 MHz. The power estimates were conducted using the FPGA vendor tools and are based on low-level timing simulations of the placed&routed designs. The estimation tool extracts switching information out of the simulation traces and applies them to a detailed, device-specific power model for the implemented design. As the device activity is lower during the initialization phase, when the matrix data is being loaded into the BlockRAM modules, we start recording the simulation traces at the beginning of the computation phase. For the simulation we use (uniformly distributed) random input sequences analogous to what we used for the in-memory computing experiments.

Table 1 presents the static and dynamic parts or the estimated power consumption. The device static power is caused by leakage on all connected voltage rails and the circuits required for the FPGA to operate normally. The dynamic power is related to charging and discharging parasitic capacitances in the circuit during operation.

Table 1: FPGA resource utilization, frequency and estimated dynamic on-chip power consumption.

LUT	FF	BRAM	f [MHz]	P_{static} [W]	$P_{dynamic}$ [W]
307908	180368	1024	200	4.04	26.6
[46.4%]	[13.6%]	[47.4%]	*(utilization on the xcku115 FPGA device)*		

The time to compute one dot-product is equal to the vector size divided by 8, plus 5 cycles to complete the pipeline. For a 1024x1024 matrix-vector product using the 1024-unit design, each dot-product unit stores one of the matrix row of 1024 elements encoded with 4-bit per value in the local 32kbit BlockRAM. To read the row vector from memory and to perform the dot-product operation takes a total of 133 clock cycles. The I/O operations to read and write the data from (to) local or host memory can be interleaved with the processing. As all of the 1024 units operate in parallel we can compute the entire matrix-vector product in the same time. Hence, it takes 665 ns to complete one matrix-vector multiplication at a clock frequency of 200 MHz. Considering a dynamic power consumption of 26.6W, one matrix-vector multiplication consumes 17.7µJ on the FPGA. The total data per vector entry is 12-bit (4-bit input, 8-bit output). For the 1024-unit architecture we need to communicate 12x1024bits (or 1.5kB) per operation which results in a total combined bandwidth requirement of 2.15 GB/s if the processing engines are running at 200 MHz.

In a memristive crossbar of size 1024x1024 based on PCM devices similar to that used in our prototype chip in 90nm technology, the dynamic power dissipation in the devices for one READ operation is expected to be on the order of 0.21W, assuming an average READ current of 1μA per device and average voltage of 0.2V. In order to operate this crossbar at 1μs cycle time, 8 analog-to-digital converters (ADCs) operating at 125MSps are needed to read the currents from all 1024 columns in approximately 1μs. The power consumption of 8-bit ADCs in 90nm technology is estimated to be around 12 mW/GSps, thus 12.3mW for 1024 reads per microsecond. Therefore, the total power consumption of the crossbar and ADCs is estimated to be around 222mW, which is 120 times lower than the 4-bit FPGA design. The energy per READ is 222nJ, which is 80 times lower than the FPGA. Those estimates show that even when neglecting the data transfers and assuming parameters based on our current prototype PCM chip, a memristive crossbar array used to perform analog matrix-vector multiplications could already offer significant energy improvements compared to low-precision CMOS-based solutions. Assuming 90nm technology and $25F^2$ 1T1R PCM cells (F = 90nm), the area occupied by a 1024x1024 crossbar and 8 ADCs (each of size 50μmx300μm) would be on the order of $0.332mm^2$.

5. CIM for IoT sensory applications

5.1 Motivation

In this section, we will explore the application of CIM architectures for the application domain of IoT sensory applications. The focus is on identifying applications that, even when implemented using low precision blocks they retain good accuracies, and whose development using CIM architectures would improve its performance/energy consumption.

5.1.1 Deep learning inference

Like the compressed sensing application, here again the computational primitive is matrix-vector multiplications using a memristive crossbar array. The forward propagation in standard Fully Connected Neural Networks (FCNN) and Convolutional Neural Networks (CNN) architectures are implemented operating with input data and a layer defined by its weights and its bias.

To map a matrix-vector multiplication to a resistive crossbar we consider:

1. The input vector [V] is represented by a vector of voltage signals and fed to the wordlines (rows) of the crossbar.
2. The matrix [G] is represented by the conductance values of the devices present in the crossbar.
3. After programming each crossbar device to the desired cell conductance, we can obtain at the bitlines (columns) the currents corresponding to the operation [I=V G].

Figure 16 represents the block of the basic primitive. We identify the DACs and ADCs involved in the analog to digital conversions that take place between different layers within the accelerator. Each ADC/DAC is characterized by a precision that determines the quantization levels of the inputs/activations of the network layer. Similarly, we can spot how, as memristive devices can only be programmed to certain conduction levels, they can represent a limited (and digital) range of weights. The quantization characteristics will determine the performance of the system and therefore its accuracy.

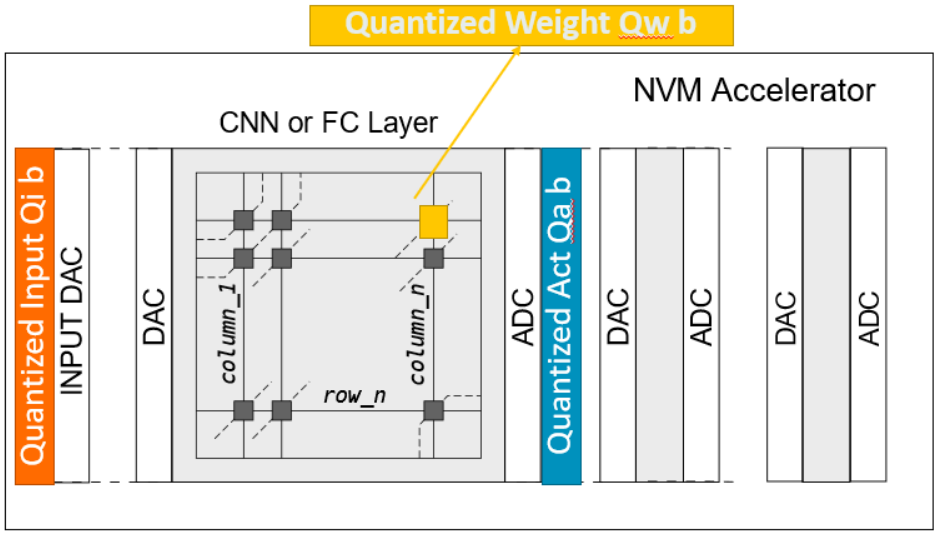

Figure 26: Basic primitive block for deep learning inference

5.1.2 IoT sensory applications

That applications that are particularly suitable for this type of an architecture are always ON inference applications that can be mapped into small/medium size crossbars and that only require limited (3-8 bits) precision. Examples include Human Activity Recognition (HAR), Key Word Spotting (KWS) and online Electro-cardiograph (ECG) event detection and classification.

- Since the memristive devices are only read, the power consumption can be very low and the limited endurance of the devices don't pose a significant challenge
- Allows always-ON applications with low power consumption
- Use of smaller crossbars reduces sneak paths and parasitic problems
- DL architectures as Fully Connected (FC) deep neural networks reduce impact of device variability and other nonlinearities
- The accuracy of the DL architecture may not be greatly affected by reduced precision or level quantization

As shown in Figure 17, the always ON CIM architectures can process the data coming from a network of sensors and sparsely wake up a higher-end CPU should a specific condition be met. This scheme allows us to continuously compute DL algorithms while reducing the power consumption.

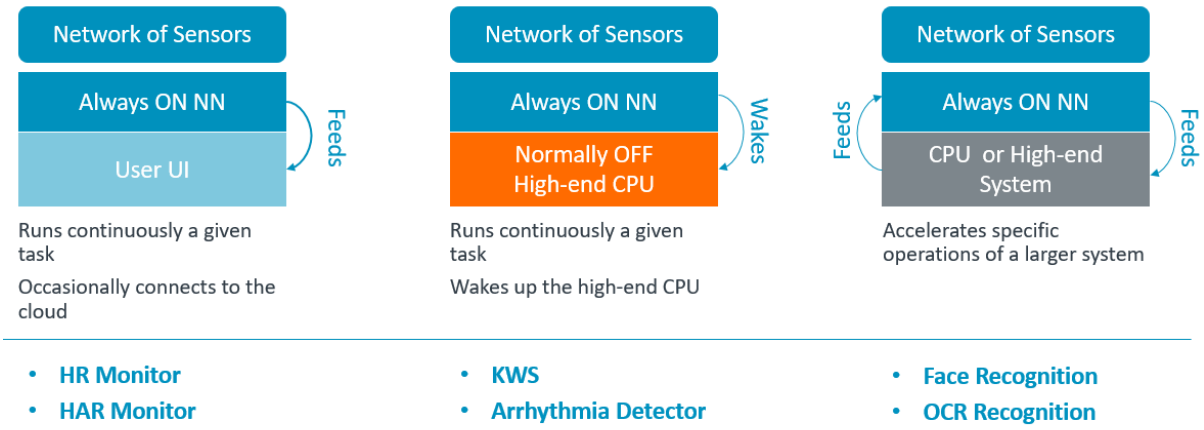

Figure 17: Different always-ON sensory applications

Figure 18 describes the proposed architecture for both applications. It is composed of several neural network layers individually mapped to memristive crossbars interconnected by ADCs/DACs. The use of ADCs/DACs allows us to isolate analog noise and non-linearities while limiting its propagation to later stages.

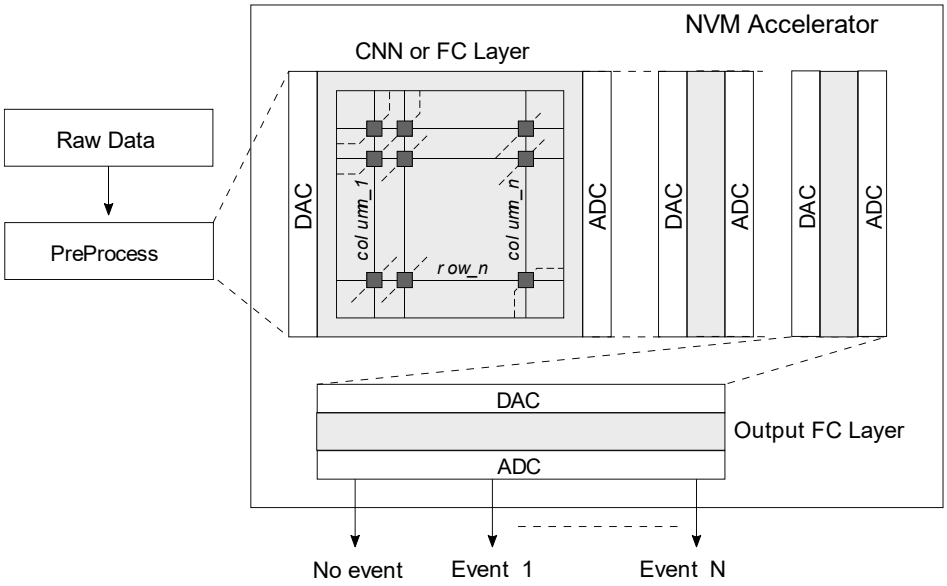

Figure 18: Proposed Architecture

5.1.3 Performance evaluation based on simulations

An ad-hoc framework was developed to systematically study the key characteristics such as the topology/number of layers, size of crossbars and required precision and to evaluate the impact caused by these design considerations on the overall application accuracy.

The framework has been developed as a wrapper that automates the computation of TensorFlow graphs. It has been optimized to determine, given requirements related to either accuracy or design constraints, a near-optimal architecture. The main characteristics of the used quantization operations can be found at [tensorflow.org/performance/quantization].

Currently, for the operations taking place during the retraining/inference stages, three different quantization schemes have been implemented:

1. Fully automated mode. A custom implementation of the TensorFlow graph quantization algorithm allows the automatic evaluation of quantization effects on a given application. This scheme is limited to the use of 2D convolutional and FC layers (and optionally BN layers).
2. Semi-automated mode for activations and weights [I]. Some activations are manually quantized during training/inference, while weights are only quantized for inference or between retraining stages.
3. Semi-automated mode for activations and weights [II]. Some activations are manually quantized during training/inference, while weights are incrementally quantized and frozen during training.

By using the proposed framework and methodology, we achieved the following initial results:

1. Application 1: **HAR**
    - Human activity recognition application processing raw signals coming from sensors present in the smartphone.
    - This application can accurately discriminate between the following activities: [walking, walking upstairs, walking downstairs, sitting, standing, laying] therefore waking up battery hungry applications running on the phone if required:
        1. Map based applications: To enable/disable GPS
        2. Physical exercise applications: To enable/disable GPS, enable/disable continuously sensing of Heart Rate HW, etc.
    - Dataset: Human Activity Recognition Using Smartphones Dataset [uci_har].
    - Small networks (0/2CNN + 2/4 FC layers, 128 to 64 neurons per layer) with low precision (8 bits inputs, 4-6 bits per weight/activation), achieving accuracies over 90%.
2. Application 2: **KWS**
    - 85-88% accuracy on classification of 12 different labels using simple networks (3 FC layers, 128 neurons per layer) and 7-8 bits to quantize weights and activations.
    - Some stages within MFCC features extraction may be accelerated in RS crossbars.
    - Expected accuracy with default configuration: 88%
        1. [Google Speech Dataset] (v0.2).
        2. 8b input quantization
        3. 8b weights quantization
        4. 8b activations quantization

5. NN: Input [250] + 3 FC layers [128 neurons each] + Output [12]

- Direct energy comparison between RS based approach and near-threshold ad-hoc Cortex-M CPU.

3. Application 3: **ECG [On-going work]**

- Processing of raw ECG signals for the detection and classification of normal beat vs 4 types of arrhythmia.
- Trained and tested against open ECG database PhysioNet [MITdb]. Currently working to integrate various datasets as PyhisioNet [PDB].
- Small networks (0/2CNN + 2/4 FC layers, [128 to 64] neurons per layer) with low precision (8 bits inputs, 4-6 bits per weight/activation), on-going work to achieve state of the art results.
- Issues: Reducing false negatives on the trained network lead to a larger number of false positives. Working on reducing the number of false positives.

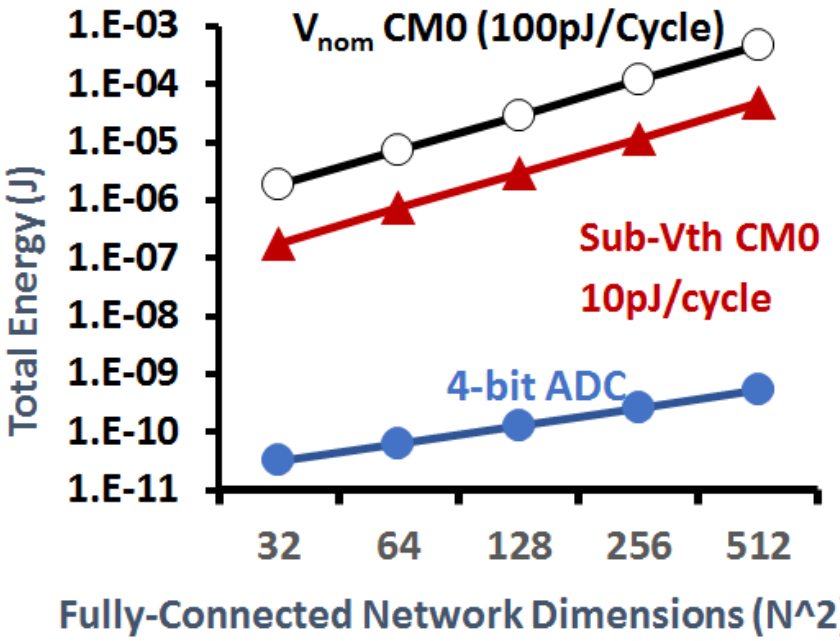

Figure 19: Preliminary comparative study between implementations using CIM architectures and on Cortex-M processors

Preliminary comparative study of implementations of the DL algorithms was conducted. The CIM approach was compared again implementations using low-power near threshold Cortex-M processors. The study shows the significant potential for energy gains with the use of a CIM architecture (see Figure 19).

6. Hyperdimensional Computing

In this section, we present another application space for CIM namely hyperdimensional (HD) computing. In HD computing formalism, information is represented in hypervectors: *d*-dimensional holographic (pseudo)random vectors with independent and identically distributed (i.i.d.) components. When the dimensionality is in the thousands, e.g. *D* >1000, there exist a

very large number of quasiorthogonal hypervectors. This lets HD computing combine such hypervectors into a new hypervector using well-defined vector space operations. These mathematical operations are bitwise and ensure that the resulting hypervector has the same dimensionality—i.e., fixed-width. The resulting hypervectors can then be directly used to not only classify but also to bind, associate, and perform other types of "cognitive" operations in a straightforward manner.

6.1 Problem description

Here, we primarily investigate the use of in-memory computing techniques to efficiently run HD computing classification tasks using phase change materials (PCM). The behavior of PCM is stochastic and subject to drift over time. Because in-memory computations are mostly analog and thus less tolerant to noise, these effects make it challenging to perform accurate computations with these devices. Hence, we attempt to understand effects of stochasticity and drift on classification accuracy of HD computing. We further propose techniques to improve the accuracy. These techniques are implemented and tested on a prototype PCM chip to validate the effectiveness of proposed techniques. Concretely, we have targeted three different applications having various discrete and analog inputs:

1. Language recognition that presents a database of texts classified under 21 European languages and offers 21000 short sentences covering same languages. These two datasets together used to apply HD computing for language recognition task.
2. News categorization that provides a collection of news articles covering 8 different genres and hence used to form a dataset for a news classification problem.
3. Electromyography (EMG)-based hand gesture recognition from five different gestures

6.2 HD Operations

HD computing uses three operations: addition (which can be weighted), multiplication, and permutation (more generally, multiplication by a matrix). "Addition"' and "multiplication"' are meant in the abstract algebra sense where the sum of binary vectors [A + B + …] is defined as the componentwise majority function with ties broken at random, the product is defined as the componentwise Exclusive-Or (addition modulo 2, $\oplus$), and permutation (ρ) shuffles the components. All these operations produce a *D*-bit hypervector.

The usefulness of HD computing comes from the nature of the operations. Specifically, addition produces a hypervector that is similar to the argument hypervectors (the inputs) whereas multiplication and random permutation produce a dissimilar hypervector; multiplication and permutation are invertible, addition is approximately invertible; multiplication distributes over addition; permutation distributes over both multiplication and addition; multiplication and permutation preserve similarity, meaning that two similar hypervectors are mapped to equally similar hypervectors elsewhere in the space.

Figure 20 shows how HD computing can be used for an application of language recognition:

1) The input letters are embedded into a HD space using what is so-called an item memory that is essentially assigning a pseudo-orthogonal hypervectors to each input letter.
2) The resulting *D*-bit hypervectors are further combined using the bitwise MAP operations inside an encoder. For example given the trigram "abc", the *A* hypervector is rotated twice ($\rho(\rho(A))$), the *B* hypervector is rotated once ($\rho(B)$), and there is no rotation for the *C* hypervector. Componentwise XOR are then applied between these

three hypervectors to compute the trigram hypervector, i.e., ρρ(A) ⊕ ρ(B) ⊕ C. This results in a *D*-bit trigram hypervector that is used for learning and classification in associative memory. During training when the language of the input text in known, we refer to this hypervector as a *text* hypervector. When the language of a text is unknown, as it is during testing, we call the hypervector a *query* hypervector.

3) During training, associative memory adds all text hypervectors to create a language hypervector representing profile of the language of interest. This process is repeated for all languages, in our case 21 times. During classification, the associative memory find the closest language hypervectors to the input query hypervector using Hamming distance, or dot product.

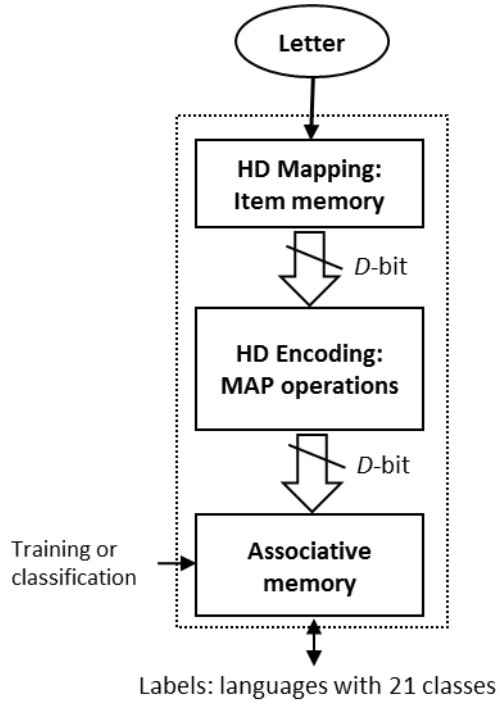

Figure 20. HD computing data flow for language recognition.

6.3 Implementation with CIM

The CIM primitives used for HD computing implementation are dot-product and bit-wise operations. The dot-product is performed using binary input values, binary memristor states, and analog output. The bit-wise operations are performed using binary input values, binary memristor states, and binary output. The memristor values are written only once before the execution of the HD algorithm and are never modified again. Additional digital computations and memory buffers are needed in order to implement the entire HD algorithm. Further implementation details are omitted due to pending patents.

6.4 Performance evaluation from simulations

Successful classification on PCM device model under low signal-to-noise ratio (SNR) conditions: matching accuracy ~3% compared to ideal software model among 3 applications.

	Software	**Software with approximation**	**Statistical PCM model simulations**
Language recognition	97.0%	96.2%	95.5%
News categorization	92.4%	91.7%	91.7%
EMG-based hand gesture recognition	98.5%	98.0%	98.0%

6.5 Energy analysis

Preliminary results were obtained comparing the energy efficiency of a potential CIM-based implementation over 65nm digital CMOS implementation. A cycle-accurate RTL model that has equivalent throughput to that of the proposed CIM HD processor was developed. The RTL model was synthesized in UMC 65nm technology node using Synopsys Design Compiler. Energy estimation was carried out in Synopsys Primetime tool by providing the netlist and the activity file as the inputs. To compare performance of baseline CMOS HD processor against the proposed CIM HD processor, we formulated the energy consumption of the CIM approach based on the measurements from PCM experiments summarized in Table 2.

	CMOS	PCM worstcase	PCM bestcase
technology	65nm	90nm	90nm
(read) voltage	1.2V	0.2V	0.1V
readout time		1us	0.1us
readout current		3uA	1uA
cell size		50F^2	25F^2

Table 2: Device parameters used in simulation.

Comparison of performance results of CMOS over PCM-based CIM are given in Figure 12 for language classification problem. We consider the best case and the worst case conditions for PCM to understand the whole range of improvements. In summary a best area improvement of 9x and an energy improvement of 5x is expected with the CIM HD processor architecture compared to CMOS counterpart. Best-case energy improvement is restrained due to energy of on-chip ADCs. By utilizing more efficient ADCs the performance numbers could be improved further. Nevertheless if only the replaceable module in the architecture are considered vast improvements can be expected which are eclipsed by the current energy budget of the non-replaceable modules. When only replaceable modules are considered, energy efficiency can be two to three orders of magnitude higher in the case of PCMs as shown in Figure 21.

	AM Search	Encoder	Total
System-level Energy (nJ)			
CMOS (per query)	1340	1780	3126
PCM worst case (per query)	42.2	576	618.2
PCM best case (per query)	3.30	581.8	585.10
Improvement best case	406.1	3.07	5.34
Replaceable Modules Energy (nJ)			
CMOS (per query)	1339	1210	2549
PCM worst case (per query)	39.6	348	387.6
PCM best case (per query)	0.66	5.80	6.46
Improvement best case	2028.8	208.6	394.6
System-level Area (um^2)			
CMOS	2916000	573000	3489000
PCM worst case	119100	393700	512800
PCM best case	74550	284350	358900
Improvement best case	39.1	2.02	9.72

Q

Figure 21: Energy comparison between 65nm digital CMOS and CIM PCM-based implementations.

7. Conclusions

In this report, we have identified several applications where a computation-in-memory (CIM) architecture would be beneficial. The computational primitives that can be implemented in a CIM can be broadly classified into two. 1) Logical operations that exploit the non-volatile binary storage capability of memristive devices 2) Matrix-vector multiplication operation that exploit the non-volatile multi-level storage capability of memristive devices. In Sections 2 and 3 we present several applications where logical operations performed in-place in the memory array can be used to increase the computational efficiency significantly. These include database query, security encryption, automata processing and image processing. In sections 4 and 5, we address applications where the efficient implementation of matrix-vector multiplication is used. These include compressed sensing and recovery and deep learning inference for always ON IoT sensory applications. We have also performed a thorough investigation of the specifications and requirements for these applications to be competitive against conventional approaches. Even though some of these applications are investigated in greater detail than the rest, over the course of the project, all of them will be studied in greater detail and more applications will be added to the list.