MOL-based In-Memory Computing of Binary Neural Networks

Khaled Alhaj Ali, Amer Baghdadi, Elsa Dupraz, Mathieu Léonardon, Mostafa Rizk, Jean-Philippe Diguet

▶ **To cite this version:**

Khaled Alhaj Ali, Amer Baghdadi, Elsa Dupraz, Mathieu Léonardon, Mostafa Rizk, et al.. MOL-based In-Memory Computing of Binary Neural Networks. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2022, 30 (7), 10.1109/TVLSI.2022.3163233 . hal-03659297

HAL Id: hal-03659297

https://imt-atlantique.hal.science/hal-03659297

Submitted on 9 May 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MOL-based In-Memory Computing of Binary Neural Networks

Khaled Alhaj Ali, Amer Baghdadi, Elsa Dupraz, Mathieu Léonardon, Mostafa Rizk and Jean-Philippe Diguet

***Abstract*—Convolutional neural networks (CNN) have proven very effective in a variety of practical applications involving Artificial Intelligence (AI). However, the layer depth of CNN deepens as user applications become more sophisticated, resulting in a huge number of operations and increased memory size. The massive amount of the produced intermediate data leads to intensive data movement between memory and computing cores causing a real bottleneck. In-Memory Computing (IMC) aims to address this bottleneck by directly computing inside memory, eliminating energy-intensive and time-consuming data movement. On the other hand, the emerging Binary Neural Networks (BNN), which is a special case of CNN, shows a number of hardware-friendly properties including memory saving. In BNN, the costly floating-point multiply-and-accumulate is replaced with lightweight bit-wise XNOR and popcount operations. In this paper, we propose an IMC programmable architecture targeting efficient implementation of BNN. Computational memories based on the recently introduced Memristor Overwrite Logic (MOL) design style are employed. The architecture, which is presented in semi-parallel and parallel models, efficiently executes the advanced quantization algorithm of XNOR-Net BNN. Performance evaluation based on CIFAR-10 dataset demonstrates between $1.24\times$ to $3\times$ speedup, and 49% to 99% energy saving compared to state-of-the-art implementations, and up to 273 image/sec/Watt throughput efficiency.**

***Index Terms*—Convolutional neural networks (CNN), Binary neural networks (BNN), In-memory computing (IMC).**

I. Introduction

Deep convolutional neural networks (CNN) are the current state-of-the-art for many computer vision tasks such as image classification, detection, and localization [1], [2]. Specifically, there is an increasing focus on the deployment of CNN in mobile systems, IoT devices and embedded chips for the mass market [3]. The main challenge that limits the integration of CNN in such systems is the requirement for a substantial amount of computation and memory. For instance, the VGG-19 network exhibits over 140 million floating-point (FP) parameters and requires more than 15 billion FP operations in order to classify one image [4]. Embedding such networks in traditional cores that deploy Von-Neumann model (e.g. CPUs and GPUs) poses significant problems in terms of execution speed and power consumption. Massive intermediate data are produced during CNN execution revealing intensive I/O data congestion between memory and processing cores causing a real bottleneck.

K. Alhaj Ali, A. Baghdadi, E. Dupraz, M. Léonardon and M. Rizk are with IMT Atlantique, Lab-STICC, UMR CNRS 6285, F-29238 Brest, France. M. Rizk is also with School of Engineering, International University of Beirut, and Physics Department, Faculty of Sciences, Lebanese University, Lebanon. J. Diguet is with IRL CROSSING CNRS, Adelaide, Australia. (e-mail: khaled.alhaj-ali@imt-atlantique.fr)

Various prior works have been proposed to alleviate the hardware burdens in Von-Neumann model in order to get better CNN inference performance. Two of the most promising solutions are network binarization [5–7] and in-memory computing (IMC) [8]. Network binarization or Binary Neural Networks (BNN) quantize all the weights and/or inputs to +1 and -1, providing a promising solution to mitigate storage and computation bottlenecks. In the resulting BNN, each convolution is processed by simple bitwise operations (XNORs and popcounts) instead of the multiply-and-accumulate (MAC). While BNNs are compact and efficient for resource-constrained devices, a degradation in accuracy is inevitable compared to their full precision counter-parts. However, recent works have been carried out to reduce the decline in accuracy [5]. For instance, the authors in [9] have demonstrated only 3% loss in accuracy when applying BNN to the CIFAR-10 dataset. For the larger ImageNet dataset, the authors in [10] have achieved promising results where the accuracy loss is around 5%.

On the other hand, IMC is one of the emerging techniques that address the memory wall problem encountered in conventional Von-Neumann model [11], [12]. By merging processing cores and the memory component into a single unit, IMC allows to perform a part of the computation inside the memory, thus eliminating the need for data exchange. Although IMC is an old concept [8], it has been revisited recently with the advent of emerging Non-Volatile Memory (NVM) technologies where computing is efficiently enabled on the storage cells, directly on the data location. Several recent IMC architectures [13–15] have been developed based on NVM technologies such as resistive memory (RRAM), magnetic memory (MRAM) and phase change memory (PCM). Usually, IMC breaks arithmetic tasks into elementary logic operations that are successively executed within the memory cells. Although IMC can execute any arithmetic task, some tasks may be more efficiently performed in classical CMOS implementations. In this case, dedicated near-memory units are usually added to handle such tasks. More recently, a computational memory (CMEM) architecture [13] based on Memristor Overwrite Logic (MOL) design style has shown promising performance in terms of execution speed and throughput efficiency especially for bitwise application tasks.

In fact, there is a great synergy between in-memory computing and BNNs when they are combined: the low logic complexity of BNNs makes them well suited for in-memory implementation. In this context, we propose a novel MOL-based in-memory architecture design dedicated for binary neural networks. The related contributions can be listed as follows:

- The proposed architecture efficiently implements a parallel row-wise in-memory XNOR-based convolution.
- A novel mechanism for combining in-memory and near memory execution is proposed for certain operations such as popcount and max pooling.
- An original in-memory bubble sorting technique is introduced to execute a majority-binarization stage replacing addition and normalization operations.
- For evaluation, a python-based environment with appropriate library and commands has been developed emulating the functionality of the adopted MOL-based computational memory and the corresponding control unit.
- The new architecture is presented in the form of both semi-parallel and parallel models, and exhibits the lowest energy consumption compared to all existing relevant works including CPU, GPU and FPGA.
- The proposed parallel model reveals the lowest inference latency when compared to recent NVM-based approaches thanks to the high level of parallelism offered using MOL-based in-memory computing.

The rest of this paper is organized as follows. Section II provides an overview on CNN, BNN and the adopted MOL-based in-memory computing approach. Section III illustrates the devised methodology and algorithms for realizing a BNN inside the CMEM architecture. Section IV describes our proposed semi-parallel and parallel architectures that are dedicated for BNNs. Section V presents the environment setup for performance evaluation and discusses the achieved results. Finally, Section VI concludes the paper.

II. Preliminaries

This section briefly reviews the basics of CNN and BNN. In addition, it introduces the MOL-based in-memory computing technique which is adopted in this paper.

A. CNN

A CNN is a particular type of neural network. It usually takes an image at the input and computes the probabilities that the image features belong to one of the output classes. Typically, a CNN consists of several convolutional and pooling layers followed by fully-connected layers (FC) as illustrated in Fig. 1(a). It has been shown that fully-connected layers could be equivalently replaced by convolutions [16].

1) Convolutional layers: As depicted in Fig. 1, a convolutional layer takes an input feature map (Ifmap) represented by a set of channels/matrices and convolves them with a particular set of weights (called kernels) to generate an output feature map (Ofmap). The transfer from Ifmap to Ofmap follows expression (1).

$$Y_m = f\left(b + \sum_{n=1}^{N} X_n * W_{n,m}\right) \tag{1}$$

In this expression, X_n represents an Ifmap channel of index n (where $n \in [\![1, N]\!]$) and Y_m represents an Ofmap channel of index m (where $m \in [\![1, M]\!]$). M and N are the number of channels of the Ifmap and Ofmap, respectively. $W_{n,m}$ is a $k \times k$ weight filter window linking X_n with Y_m. The parameter b is the bias. f represents the activation function.

2) Pooling layers: Pooling is an important feature of CNN as it reduces the dimensionality of a feature map while maintaining the most important information [17]. It allows to reduce the size of the network and the number of parameters used, preventing overfitting. Considering the max pooling, a spatial neighborhood (e.g. a 2×2 window) is defined. The window is slided without overlapping on the Ofmap elaborated by the convolutional layer. The largest element inside that window is taken as an output. Another choice is to take the average (Average Pooling) or the sum of all elements in that window. In practice, max pooling has been shown to work better [17]. An intuitive example of max pooling is illustrated in Fig. 1(b).

B. BNN

The multiply-accumulate is the key and the most computationally expensive arithmetic operation in classical CNN. BNNs have been introduced to alleviate the need for these operations. This is achieved by forcing the inputs/weights/gradients to have binary values, especially in the forward propagation.

Various types of BNNs have been explored in the literature [9], [10], [18]. In this paper, we adopt the XNOR-Net [18] binary neural network which offers significant simplifications and better results than other binarization methods. In XNOR-Net, both the incoming activations and weight parameters of the convolutional layers are constrained to a binary set {-1,1} except for the first convolutional layer where the input is the image. For efficient hardware mapping, the values -1 and 1 are encoded to logic '0' and '1' respectively. Then, multiplication of weights and activations is achieved according to the XNOR truth table as shown in Fig. 2(a).

As illustrated in Fig. 2(b), the accompanied MAC operations can be then replaced by a series of XNOR operations and a final popcount (difference between number of zeros and ones). The result is then subjected to normalization and binarization (Norm-bin). The convolutional layer in a XNOR-net BNN is depicted in Fig. 3 and can be modeled by the following expressions:

$$y^b_{n,m} = \text{Norm-bin}\left(\text{Popcount}\left(XNOR\left(W^b_{n,m}, X^b_n\right)\right)\right) \tag{2}$$

$$Y^b_m = \text{Norm-bin}\left(\sum_{n=1}^{N} y^b_{n,m}\right) \tag{3}$$

where $y^b_{n,m}$ represents the output after convolving the n^{th} binary Ifmap X^b_n with its corresponding binary weight kernel $W^b_{n,m}$, and Y^b_m represents the m^{th} Ofmap after adding and binarizing all the N outputs $y^b_{n,m}$.

C. MOL-based in-memory computing

1) In-memory computing (IMC): IMC has been widely explored to overcome the memory wall by avoiding the long latency originated from intensive exchange of data between host processor and memory. From the device level perspective, emerging Non-Volatile Memories are promising for the implementation of IMC. In this context, several recent contributions have been proposed and can be classified in two categories. The first category includes approaches that use the NVM cell

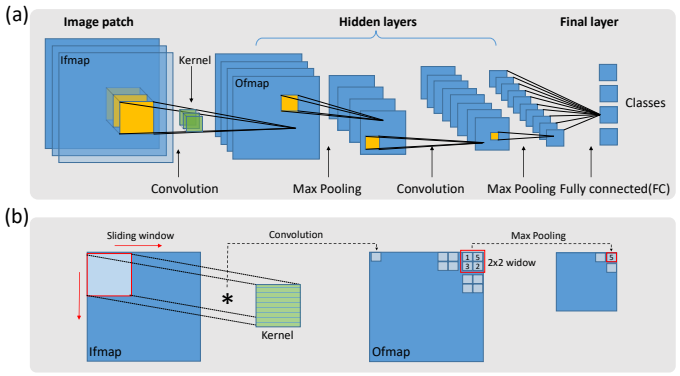

Fig. 1. The structure of the convolutional neural network: (a) the multiple layers of CNN including convolution layer (CONV), pooling (POOL) and fully-connected (FC) layer, and (b) illustration on the convolution and the Max Pooling operation.

(a)

XNOR truth table

Encoding (Value)		XNOR (Multiply)
0 (-1)	0 (-1)	1 (+1)
0 (-1)	1 (+1)	0 (-1)
1 (+1)	0 (-1)	0 (-1)
1 (+1)	1 (+1)	1 (+1)

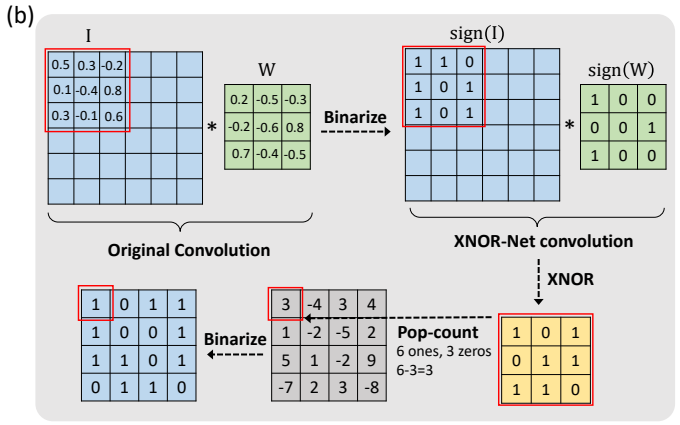

Fig. 2. Binary Neural Network: (a) XNOR truth table, and (b) XNOR-popcount process.

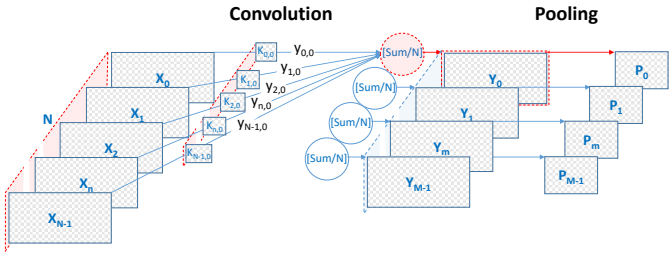

Fig. 3. A convolutional layer in a BNN followed by a pooling layer

as a single-level cell (SLC) [13–15]. In the second category, the authors have employed the NVM cell as a multi-level cell (MLC) or analog cell [19–21]. MLC crossbars can perform parallelized in-situ operations by eliminating sequential memory accesses. MLC-based computing is promising when targeting applications with intensive MAC operation (e.g. CNN) [22]. However, a number of challenges remain in terms of manufacturability and computational accuracy regarding device variability, pattern-dependent current leakage and the area overhead of peripheral circuits [23]. In contrast, SLC approach involves larger readout margin that makes NVM cells much tolerant against process variation and resistance drift effects.

Based on the SLC approach, various in-memory computing techniques have been introduced in the literature [14], [15]. All these techniques attempt to realize arithmetic tasks inside NVM arrays by performing successive elementary logic operations on the stored data bits. For instance, the Material Implication (IMPLY) [15] and the Memristor Aided Logic (MAGIC) [24] have been introduced to enable in-place logic operations in memristive crossbar arrays. Although promising results have been demonstrated, these techniques present the following limitations:

- The performance of IMPLY and MAGIC is highly dependent on the technology of the adopted NVM device (e.g. requirement of high ON-OFF margin) [14][15]. Thus, they are not qualified for the operation with spintronic devices such as STT-MRAMs.
- The analysis in [25] reveals that IMPLY may incur partial switching and significant state drift issues [14][15] of the NVM devices within the memory array.
- The corresponding basis functions provided by IMPLY and MAGIC are not diverse enough to allow fast logic mapping with minimum number of computational cycles.

Other in-memory computing techniques such as the sensing-based computing, that is introduced in [26], has gained large interest for its ease of implementation and the ability to execute diverse types of bitwise operations. Sensing-based computing redesigns the read circuitry so that it can compute the bitwise logic of two or more memory rows. Although fast, this technique involves a relatively high precision read circuitry that is based on sensing amplifiers (usually Op-Amps) employed as comparators. The read circuitry which must be activated at each computational step involves a relatively high energy consumption.

In this paper, the recently proposed in-memory computing approach [13] namely Memristor Overwrite Logic (MOL) is adopted. The main idea behind MOL was to address the aforementioned limitations. MOL applies for various NVM technologies spanning resistive devices (memristors [27]), spintronics (STT-MTJ [28]) and phase-change materials (PCM [29]). Moreover, it allows for significant reduction in the number of required computational steps as well as the reserved processing area inside the memory.

2) MOL-based computational memory (CMEM): The computational memory presented in [13] is composed of two adjacent non-volatile (NV) sub-arrays that work in complementary manner. In this architecture, two wordlines are activated

simultaneously in order to perform bitwise logic operations. A vector level AND/OR operation can be executed within a single computational step. Moreover, shifting and inversion operations are also enabled through a dedicated intermediate driver offering flexibility in the operations, which is crucial for some arithmetic tasks. The intermediate driver involves a simple sensing circuitry followed by a set of 2-to-1 multiplexers. The role of multiplexers is to either pass the value from the sensed wordline, or to pass its inverse. It can be configured depending on the desired operation. The multiplexers are followed by a configurable 1-bit shifter, that can be enabled/disabled during the run time. The resulting intermediate driver passes the signals on-the-fly without additional computational steps. More details on the working mechanism of the intermediate driver can be found in [13].

An external controller arranges the operations performed inside the memory sub-arrays in order to finalize a desired arithmetic task. It breaks down the task into series of micro-operations (bitwise MOL) that are executed one after the other. The architecture diagram is presented in Fig. 4.

Generally, the high bandwidth of in-memory computing allows to minimize the step period of each micro-operation. However, when the complexity of an arithmetic task scales up, the corresponding number of computational steps becomes large which again increases latency and energy consumption. In this case, moving data to be processed near memory is more efficient. Hence, the performance of in-memory computing is task dependent.

In this paper, we consider applying the principle of MOL and the corresponding computational memory for the implementation of BNNs. For higher precision neural networks, as previously stated, the multiply-accumulate operation is the crucial operation in these networks. Addition and multiplication tasks can be executed in-memory, but usually necessitate large number of computational steps which grows when increasing the size of operands (i.e., precision of the network). As a result, such tasks might be more efficient to implement in traditional Von-Neumann architectures, despite the high communication cost between memory and processing cores. In contrast, the low logic complexity of BNN makes it highly suitable for in-memory computing as the multiply-accumulate operations of a convolutional layer are mainly replaced with basic bit-wise XNOR and popcount.

III. MOL-based in-memory BNN

A. In-memory XNOR-based convolution

1) Method: The bitwise XNOR represents the most computationally expensive operation in the convolution process. Thus, optimizing its execution inside memory will enhance the overall performance of the binary neural network. This is why we first propose an efficient implementation of the XNOR convolution inside the CMEM. We consider a binary Ifmap X^b of size $h \times w$ being XNORed with a $k \times k$ weight kernel W^b. A simple example presented in Fig. 5(a) illustrates some of the steps of the proposed procedure. In this example, a 3×3 kernel is adopted. This kernel size is also suitable and used for large networks and datasets [30].

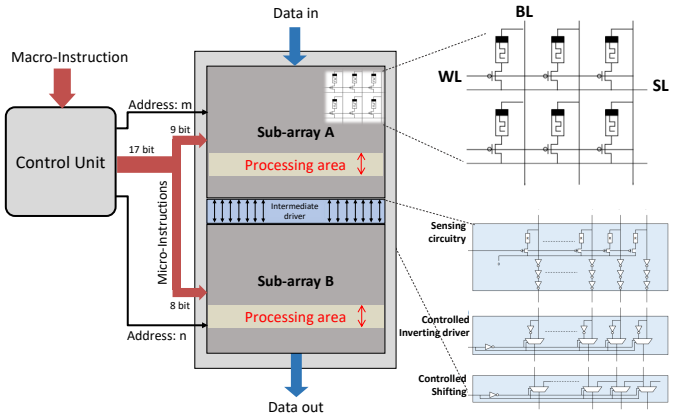

Fig. 4. Architecture diagram of the MOL-based Computational Memory (CMEM)

First, an Ifmap, zero padded at its surrounding, is loaded into sub-array A of the CMEM as shown in Fig. 5(a). The corresponding kernel is then loaded into sub-array B, though in a tiled pattern fitting the width of the Ifmap. Hence, the size of the tiled kernel is $k \times w$. Here we assume that the width after zero padding is a multiple of k, otherwise padding is increased. For better scheduling of the XNOR operations and in order to obtain a higher level of parallelism, the Ifmap is gridded into $k \times k$ slots without overlapping. For simplicity, we use the term sliding grid (instead of sliding window) to point on the selected regions that would be XNORed. The convolution is performed in k successive phases (3 phases in this example), while each phase is carried out in k rounds. For a given phase $i \in [\![0, k-1]\!]$ and round $j \in [\![0, k-1]\!]$, the sliding grid is right and down shifted to the position i and j respectively. Simultaneously, the tiled kernel is right shifted to the position i. In other words, the horizontal movement of the tiled kernel follows that of the sliding grid. This specific shifting operating is enabled by the CMEM. Row-wise XNOR is then performed purely inside the CMEM based on a series of MOL operations that are discussed in the next sub-section. At each position (i, j) of the sliding grid, a XNOR matrix is obtained in sub-array-B. The resulting matrix is then moved to a near memory processing unit to undergo popcount and binarization. In fact, the popcount process is not adapted to in-memory computing. Its implementation is inefficient in terms of the number required computational steps. Fig. 5(b) depicts the occupied regions inside the CMEM after execution, where auxiliary processing regions are required in sub-array-A and B in order to finalize the convolution task.

2) Row-wise XNOR: The Ifmap undergoes several rounds of XNOR operations inside the CMEM. Thus, an Ifmap should remain safe until the end of the convolution process. Similarly, a safe version of the kernels should be available all the time in order to be reused for later inference. Within these requirements, a non-destructive row-wise XNOR is applied based on a series of MOL operations. As illustrated in Fig. 6(a), an Ifmap row X^b_r and a kernel row W^b_r are assumed to be initially inside the CMEM sub-arrays A and B respectively. In order to avoid destroying W^b_r, a copy is stored in the processing

region of sub-array B. After that, 5 successive computational steps are required. In each step, a micro-operation $z@(m,n)$ is sent to the CMEM, where z corresponds to one of the 30 micro-instructions described in [13], m and n represent the corresponding addresses of sub-arrays A and B respectively. The 6 micro-operations are defined in Fig. 6(b). At the end, the value $X_r^b \oplus W_r^b$ is obtained in sub-array B while keeping X_r^b and W_r^b safe. As a result, only three auxiliary rows have to be reserved for the XNOR operation to be processed. As presented in Fig. 6(a), one row resides in sub-array-A (at address m_2), while the other two rows reside in sub-array-B (at addresses n_1 and n_2). The same auxiliary rows can serve for later XNORs. Yet, it is preferred to change the location of the processing region regularly to avoid the thermal accumulation in certain cells and maintain high endurance. Overall, XNORing an Ifmap of size $h \times w$ located in sub-array-A, with a tiled kernel of size $k \times w$ located in sub-array-B, requires a minimum storage space of $S_{XNOR} = (h+k+3)w$.

On the other hand, it is possible to retrieve the value of W_r^b logically instead of saving a spare copy. This is due to the fact that the XNOR operation can be reversed. The value of W_r^b is still contained in the XNOR result. Yet, the required operations for retrieving W_r^b value is definitely more expensive in terms of computational steps, in particular that the same set of kernels are frequently needed for multiple operations and inferences. Therefore, it is be more efficient in this case to use the copy-saving strategy.

3) Near memory popcount: The XNOR matrix obtained in sub-array B is also grouped into $k \times k$ slots using the sliding grid. Horizontal slots are then moved row by row to the popcount and binarize (PB) block located inside the Near Memory Unit (NMU) as shown in Fig. 5(c). In fact, the popcount and binarize process is equivalent to obtaining the majority bit in a given slot. Hence, the PB block counts the number of ones in the horizontal slots simultaneously using parallel adders and accumulators. The resulting number in each slot is then compared with the threshold value $(k^2-1)/2$, where this value corresponds to half of the number of elements in each slot. Exceeding this value indicates that the corresponding majority bit is '1', otherwise the bit '0' is returned. The architecture of PB is shown in Fig. 5(d). At the end of k successive steps, a binary row is returned to the memory. The accumulators are cleared before beginning with the next horizontal slots.

B. BNN layer in-memory

1) Convolutional layer: As illustrated in Section II-B, the m^{th} Ofmap Y_m^b is obtained by adding, normalizing and binarizing the resulting N convolution channels $y_{n,m}^b$ as expressed in (3). In fact, executing these three processes separately inside the CMEM is computationally expensive. In order to handle these processes efficiently by the CMEM, we introduce the Majority-Bin stage instead.

The Majority-Bin stage involves computing the majority channel out of the N convolution channels $y_{n,m}^b$. Computing the majority is simply achieved through a proposed in-memory bubble sorting technique. As illustrated in Fig. 7(a), we consider an arbitrary vector of bits $p = \{p_0, p_1, .., p_i, .., p_{N-1}\}$ where $i \in [\![0, N-1]\!]$. When applying bubble sorting on p, bits holding the value '1' are swept to one side of the vector leading to new vector $q = \{q_0, q_1, .., q_i, .., q_{N-1}\}$. Since the number of '0's and '1's after sorting remains the same, the middle bit $q_{N/2}$ (if N multiple of 2) can be considered as the majority bit.

The applied bubble sorting technique is implemented using the logic tree presented in Fig. 7(b), which has a regular form and is based on the proposed bubble-flip module illustrated in Fig. 7(c). The module is composed of only AND and OR logic gates. It allows to flip the input bits so that '1's appear at the output port of the OR logic gate as illustrated in the figure. As an optimization step, since the goal is to compute the majority bit and not to sort the bits inside the vector, bubble-flip modules and logic gates which are not participating in the generation of the majority bit have been discarded from the logic tree. The resulting optimized logic tree has been efficiently realized inside the CMEM, where the equivalent MOL operations are implemented sequentially.

Moreover, operations inside the CMEM are performed on the vector level. Thus, channels sorting is achieved row-wise, speeding up the whole sorting process. Fig. 7(d) depicts how rows are being sorted ending up with a majority channel Y_m^b located in the middle. The required number of steps to generate a majority channel in-memory is derived as $S_{maj} = h(\frac{3}{2}N^2 - 4N + 3)$. The derived expression shows that the latency overhead of the sorting technique is independent of the width w of the channels. Although the number of steps grows quadratically when increasing the number of channels to be sorted, the high amount of parallelism at the vector level and CMEM banks level is sufficient to compensate the time overhead of the sequential sorting process.

In fact, the concept of sorting has already been adopted in the literature for replacing computationally expensive tasks. In [31], the authors have adopted the bitonic sorting algorithm in order to replace the floating-point addition and activation stages in neural networks. Authors of [32] have implemented the bitonic sorter algorithm in memristive memory array achieving significant reduction in the processing time compared to prior sorting designs. Although bubble sort is slower than bitonic, it has a simple and more regular pattern for accessing the data to be sorted. This reduces the complexity of the control part of the architecture. In [33], the authors have proposed a general hardware/software design to achieve efficient sorting of floating point numbers for data ranking. However, the mechanism imposes a custom design, which is not adapted for in-memory computing of BNNs.

2) Pooling layer: The input channel to the pooling layer is gridded into 2×2 slots without overlapping. Each slot corresponds to a max-pooling window $p = \{p_0, p_1, p_2, p_3\}$. The max-pooling is defined as in [5] as: $max - pool(p) = 0$ if and only if all elements inside p is equal to '0', otherwise, $max - pool(p) = 1$. In fact, this can be handled by applying an elementary OR operation to the 4 elements inside p. The resulting bit corresponds to the $max-pool(p)$. Performing this inside memory is achieved in two stages. The first stage can be efficiently performed using in-memory MOL operations. Elementary OR operations are applied to the rows where all slots

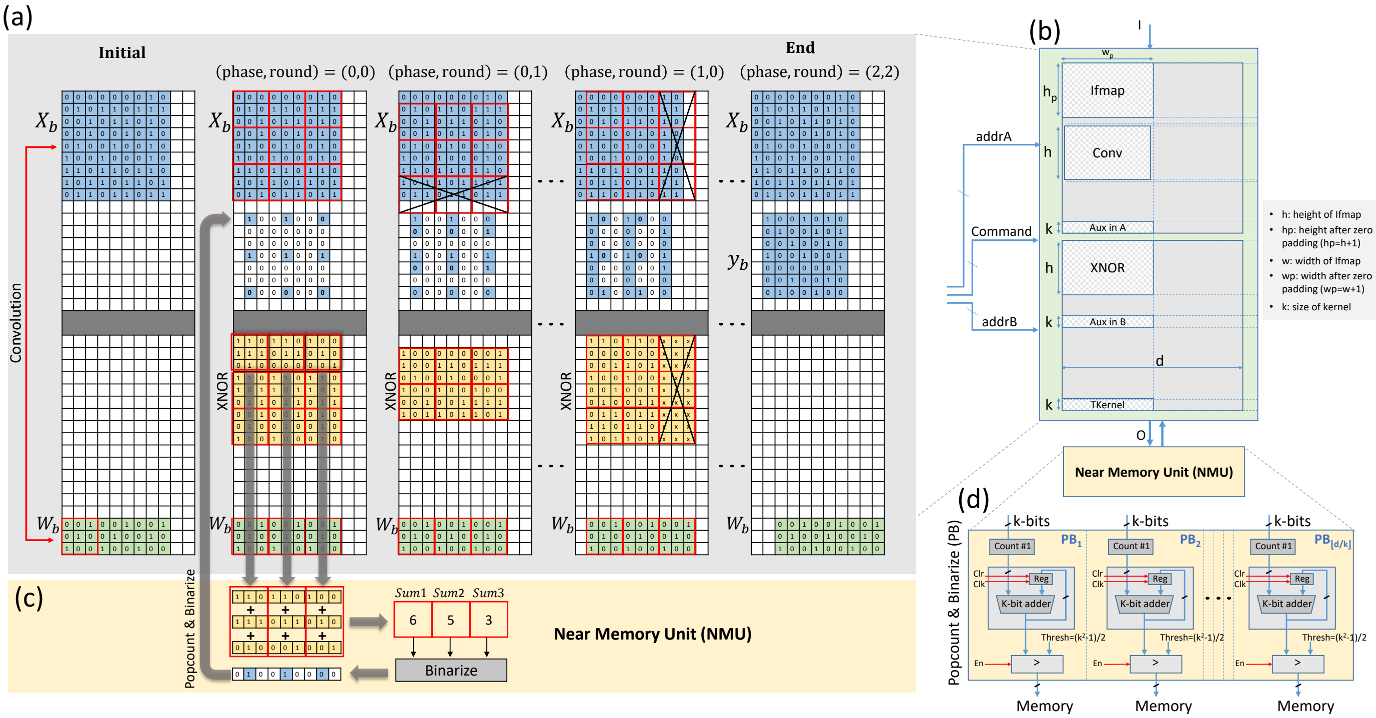

Fig. 5. The proposed in-memory XNOR-based convolution method: (a) samples of the CMEM state taken from different operation phases, (b) CMEM regions partitioning, (c) example on the near memory popcount and binarization stage, and (d) architecture of the Popcount & Binarize (PB) block located inside the Near Memory Unit (NMU).

(a)

MOL-based XNOR (6 steps)

	Command / Address	Initial	6@(m_2, n_3)	9@(m_2, n_1)	19@(m_1, n_1)	7@(m_1, n_2)	14@(m_2, n_2)	17@(m_2, n_1)
Sub-array A	m_1	X^b	X^b	X^b	X^b	X^b	X^b	X^b
	m_2		W^b	W^b	W^b	W^b	W^bX^b	W^bX^b
Sub-array B	n_1			$\overline{W^b}$	$\overline{W^b}\,\overline{X^b}$	$\overline{W^b}\,\overline{X^b}$	$\overline{W^b}\,\overline{X^b}$	$\overline{W^b}\,\overline{X^b} + W^bX^b$ (X^b ***XNOR*** W^b)
	n_2					X^b	X^b	X^b
	n_3	W^b	W^b	W^b	W^b	W^b	W^b	W^b

$M_A(m)$: *memory content in subarray-A at address m*
$M_B(n)$: *memory content in subarray-B at address n*

(b)

Command: z@(m,n)	Description	Micro-operation
6@(m,n)	Copy	$M_A(m) = M_B(n)$
7@(m,n)	Copy	$M_B(n) = M_A(m)$
9@(m,n)	Invert	$M_B(n) = \overline{M_A(m)}$
14@(m,n)	AND	$M_A(m) = M_A(m)$ AND $M_B(n)$
17@(m,n)	OR	$M_B(n) = M_B(n)$ OR $M_A(m)$
19@(m,n)	AND-NOT	$M_B(n) = M_B(n)$ AND $\overline{M_A(m)}$

Fig. 6. Row-wise XNOR: (a) operations sequence for a MOL-based in-memory XNOR operation, where each column corresponds to the state of the CMEM at a certain step, and (b) definition of the six micro-instructions used.

are managed simultaneously. Thus, the four bits in each slot are compressed into two bits placed horizontally as depicted in Fig. 8. In the second stage the resulting two bits are subjected to another OR operation resulting in the $max - pool(p)$. This stage is realized near memory using simple OR logic gates that are applied to the columns simultaneously.

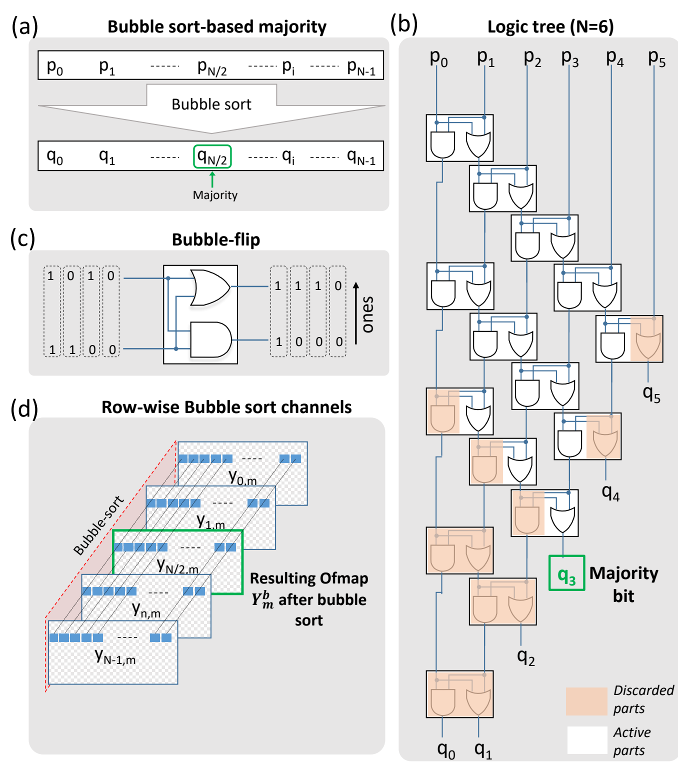

Fig. 7. Proposed bubble sort-based majority technique: (a) the middle bit after bubble sort represents the majority bit, (b) the used bubble flip module, (c) logic tree implementation for input vector size N=6 and (d) the in-memory row-wise channel sorting in order to obtain the Ofmap Y^b_m.

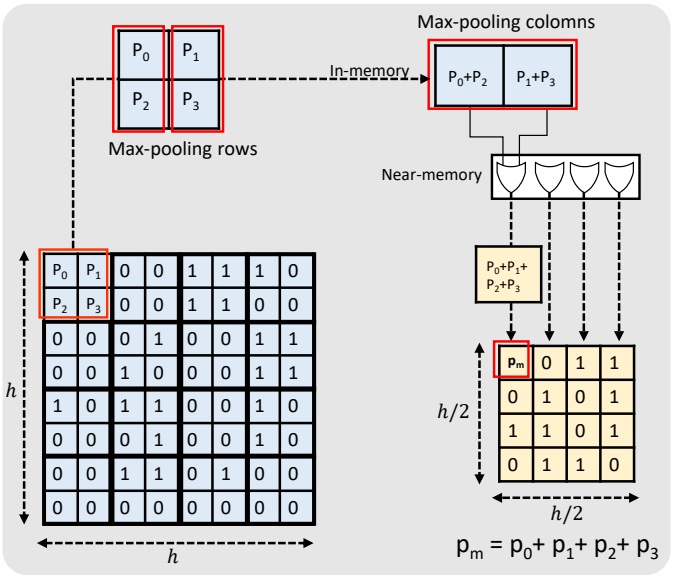

Fig. 8. Proposed in-memory & near memory combined Max Pool process.

IV. Semi-parallel/parallel architecture

This section presents our proposed architecture for the execution of BNN in-memory. Then, the control flow and the data mapping methodology are illustrated.

A. Architecture design

The execution of a complete BNN layer in a single CMEM block is time consuming, as convolutions would be executed serially. In order to improve parallelism and consequently accelerate processing, we propose to split the CMEM block into smaller interconnected CMEM units. Each CMEM unit is then employed to execute a single Ofmap channel Y_m^b in a given layer. The same units are reprogrammed to perform the other layers. The communication between these CMEM units is managed by a master memory that receives intermediate results and redistributes them in a systematic mapping method that is illustrated in Section IV-B.

Fig. 9 shows our proposed architecture with two different models. The semi-parallel model shown in Fig. 9(a) shares one NMU to all CMEM banks; whereas in the parallel model shown in Fig. 9(b), each CMEM bank reserves a separate NMU. It is worth mentioning that all CMEM units are shared with a unique control bus due to the fact that tasks being executed on different units are identical. Thus, a single control unit is able to cope with all these units. For this purpose, an enabling decoder block is added. It can be configured to either activate a single CMEM unit or all units at a time. Such an architecture model is equivalent to a single-instruction multiple-data (SIMD) model for in-memory computing, so it can be called as SIMM which stands for single-instruction multiple-memory.

B. Mapping methodology

The adopted mapping method is an important factor in our proposed architecture as it highly influences the overall performance of inferencing. We classify the mapped data depending on its static or dynamic nature. Static data, such as weight kernels, is mapped during the configuration phase only and does not require any update during the running phase. On the other hand, the continuously generated data during the running phase is considered as dynamic. This type of data requires redistribution before initializing the next layer. Examples of dynamic data are the generated convolutions and Ofmap channels.

The control flow is thus divided into three phases: configuration phase, running phase and redistribution phase:

1) Configuration Phase: As stated earlier, each CMEM unit is supposed to handle the computation of a single Ofmap channel in each layer. Thus, the m^{th} CMEM unit (CMEM$_m$) receives from the master memory its own set of weight kernels ($W_{n,m}^b; n \in [\![0, N-1]\!]$) that are required to execute the Ofmap channel Y_m^b. Moreover, the master memory broadcasts the Ifmaps to all CMEM units.

2) Running Phase: The control unit receives a flag to begin the inference process. All CMEM units are then activated simultaneously for parallel execution. For the case of the semi-parallel architecture model, popcounting and pooling stages are performed sequentially within the near memory unit. Thus, during these stages, only one CMEM unit is activated at a time. At the end of this phase, an Ofmap is generated in each CMEM unit.

3) Redistribution Phase: This phase prepares the execution of the next network layer. Normally, the generated Ofmaps are fed to the next layer as Ifmaps. They are returned one after the other to the master memory, which in turn rebroadcasts them to all CMEM units.

In fact, a single CMEM unit should have a certain minimum space to be capable of holding the inputs, the outputs and the intermediate results. For a given layer with N Ifmaps and M Ofmaps, generating a single Ofmap requires N tiled kernels. Assuming the size of Ifmaps and tiled kernels as $h \times w$ and $k \times w$ respectively, the minimum storage space required to implement the layer can be expressed as $S_{CMEM} = w(2hN + kN + 3h)$.

V. Evaluation and results

This section presents the validation and evaluation results of the proposed in-memory BNN architecture. In addition, it presents detailed discussions and comparisons with recent relevant works.

A. Simulation environment and functional validation

For functional validation, a python based environment, with appropriate library and commands, has been developed to mimic the MOL-based in-memory computing operations performed in the CMEM. This environment constitutes a proof-of-concept that illustrates the programmability of the proposed IMC architecture. It provides an abstract programming library that hides the complexity of the low-level logic implementation to application designers. In this environment, we deal with the CMEM sub-arrays as two coupled binary matrices, and each one has a separate input signal which points to a certain row

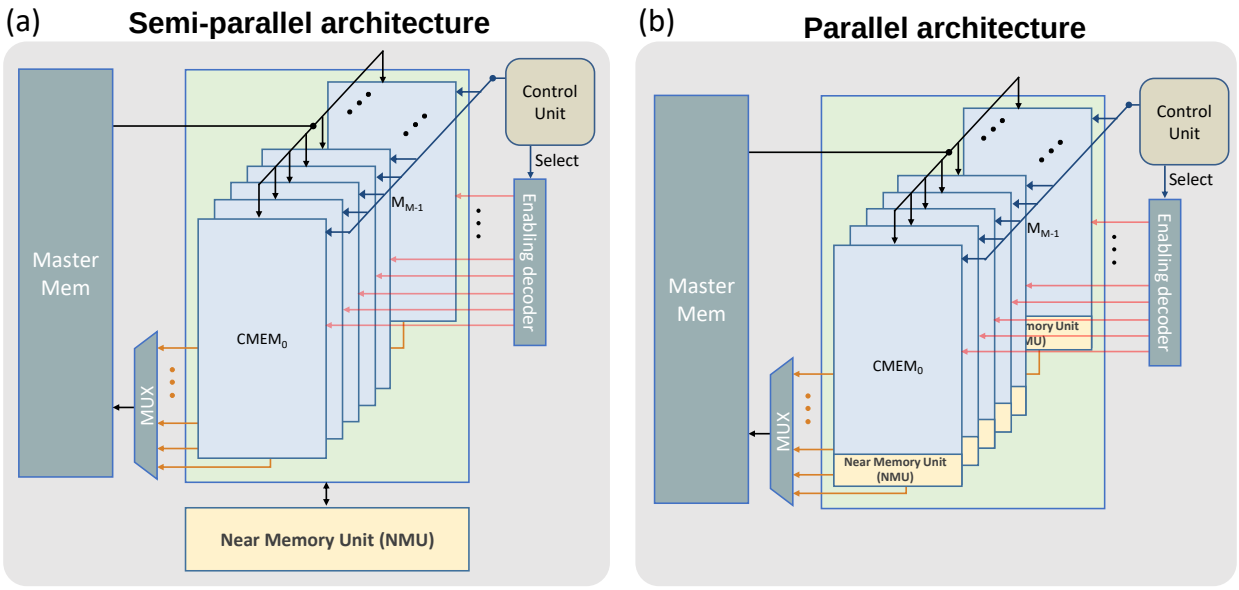

Fig. 9. The proposed in-memory BNN architecture with two different models: (a) the semi-parallel model shares one Near memory Unit (NMU) to all CMEM banks and (b) the parallel model where each CMEM unit reserves a separate NMU.

inside the matrix. The '0's and '1's of the binary matrices correspond to the ON and OFF state of the NVM cells. The two matrices receive a command which is directly translated into a bitwise elementary operation where 30 types of operations are supported and described in [13]. The developed environment allows for system-level simulations as well as performance evaluation. Within this environment, the proposed in-memory XNOR-based convolution, the Majority-Bin and pooling operations have been simulated and cross-validated with standard methods. Moreover, the implementation of binary convolutional layers has been functionally validated.

Fig. 10 shows a simple example of a BNN layer implemented with four 28×28 Ifmaps taken from MNIST dataset. The black and white pixels represent the logic states '1' and '0' of the memory cells respectively. This figure, which is extracted from the developed environment, corresponds to the state of the CMEM sub-arrays at three different instants. The first instant corresponds to the initial state, where 4 Ifmaps and a set of tiled kernels appear in sub-arrays A and B respectively. The second instant is the end of the convolutional layer, where 3 Ofmaps are generated in sub-array B. The last instant corresponds to the end of the combined in-memory & near memory Max Pooling process. The channels after rows and columns Max Pooling processes appear inside sub-arrays A and B respectively.

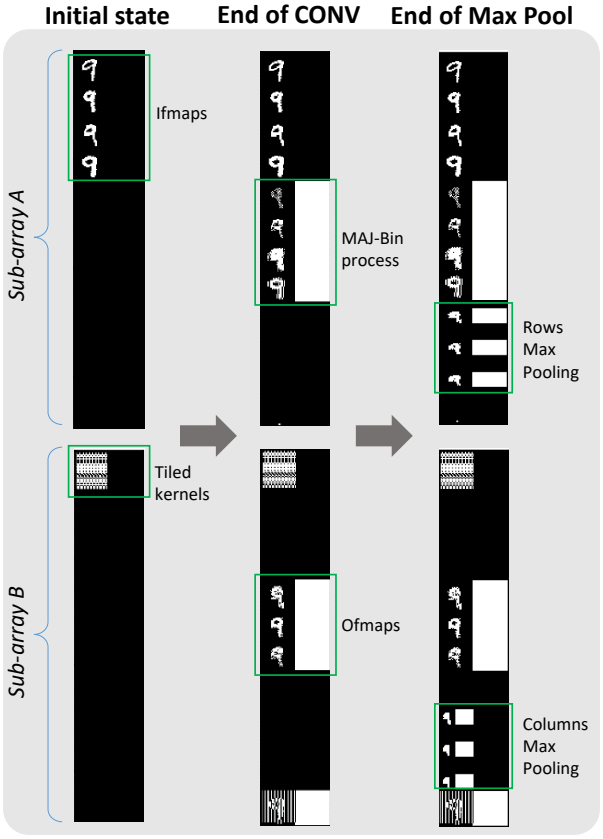

Fig. 10. The CMEM state at three different instants: the initial state, the end of the convolutional layer and the end of the Max Pooling layer.

B. Performance evaluation

In order to evaluate the performance of our proposed architecture, the binarized neural network BinaryNet [9] and the CIFAR-10 dataset have been taken as a case study. The BinaryNet model has been chosen as it achieves near state-of-the-art results on CIFAR-10 and allows comparing with the available relevant works [3], [34], [35]. For the rest of the paper, the adopted model is referred to as CIFAR-10 BNN model. The BinaryNet BNN consists of six convolutional

layers, three fully connected layers, and three pooling layers. All convolutional layers use 3×3 filters and edge padding. Both Ifmaps and weight kernels of the convolutional layers are binarized to -1 and 1 except for the first convolutional layer where the input is the image. All pooling layers employ a 2×2 max-pooling window without overlapping.

We consider the convolutional layers CONV2-5 in our evaluations. For CONV2, CONV3, CONV4 and CONV5 layers, the number of input and output channels are 128 and 128, 128 and 256, 256 and 256, and 256 and 512 respectively. The corresponding sizes of the Ifmaps are 32, 16, 16, and 8 respectively. More details on CIFAR-10 BNN model parameters can be found in [34].

For both parallel and semi-parallel architectures, we employ 128 CMEM units, each of 34-bit width. The adopted width supports the maximum binary image size (i.e. 32×32) of the CIFAR-10 after edge padding. Layers with a number of output channels exceeding 128 are performed on several stages.

The minimum step period and the average energy consumption of the performed operations are extracted from the Cadence Virtuoso toolset. The 65-nm technology node is used for the peripherals including the intermediate driver and NMUs, while a realistic spin-transfer-torque magnetic tunnel junction (STT-MTJ) device model [28] is adopted for the CMEM memory cells. The model describes the static, dynamic, and stochastic behaviours of the STT-MTJ device. It has been proven through a resistance variability analysis in [13] that a variation up to 21% in TMR, t_{sl} and t_{ox} parameters of the MTJ device leads to error-free MOL operations. In this work, we consider that the resistance variability is below this limit. According to [13], a single MOL operation consumes $0.196pJ$ of energy per STT-MTJ cell, with a maximum switching delay of $1.8ns$. The reported value of energy includes the average energy consumed by the peripherals as well as the intermediate driver. On the other hand, the switching delay corresponds to a clock frequency of 555 MHz. Since the switching speed of MTJ devices is slower than that of CMOS, this frequency is set to the entire architecture including the CMOS peripherals.

Moreover, evaluation is carried out using the emerging SOT-MTJ device which exhibits fast read/write speed and unlimited endurance. The read/write energy and delay for this device are taken from [36]. The maximum estimated values of energy ($0.1pJ$) and delay ($\sim 1ns$), for a single MOL operation, are then deduced according to the analytical expressions derived in [13]. Here, the delay corresponds to a maximum clock frequency of 1 GHz. The average energy consumed by a single MOL-type operation (AND, OR, AND-NOT, OR-NOT) per MTJ device can be expressed as $E_{MOL} = E_w/2 + E_r$, where E_r and E_w represent the read and write energy of the adopted MTJ device respectively [13]. The energy consumed by a Copy operation can be expressed as $E_{COPY} = E_w + E_r$. Other operations such as Invert and Shift have an energy overhead close to that of a Copy operation. A slight difference appears due to the change in the configuration of the intermediate driver. To evaluate the energy consumed by 34-bit width operations, the above expressions should be multiplied by a factor of 34. Other sources of energy consumption appear during operands movement from the memory array to the NMU. The NMU is located close to the memory array, thus energy consumption is mainly limited to the hybrid CMOS-resistive structure of the read circuitry, as well as the dynamic energy dissipation of the NMU's CMOS structure. Table I presents the energy consumption of the different 34-bit width operations performed in STT and SOT CMEM. All these extracted informations are provided to the developed python-based environment, which is in turn used as an evaluation tool.

TABLE I
ENERGY CONSUMED PER 34-BIT WIDTH OPERATION IN CMEM (IN PJ)

MTJ device	MOL	Copy	Invert	Shift	XNOR
STT-MTJ	6.66	11.32	11.93	12.3	54.4
SOT-MTJ	3.46	6.15	5.78	5.98	26.5

The total computational energy, power, latency and throughput efficiency for the CONV2-5 layers are evaluated in both parallel and semi-parallel architectures. Comparison is carried out with relevant recent state-of-the-art works implementing BNNs using similar in-memory computing approach [3], [35]. Moreover, BNN implementations using conventional computing approach on Intel Xeon E5-2640 processor (CPU), NVIDIA Tesla K40 GPU and FPGA [34] are considered, although the two approaches are hardly comparable. Indeed, the emerging STT/SOT memory technologies are still in the development phase and there is no mature production technology for them. In contrast, real fabricated devices using mature technologies are available for accurate performance measurement on CPUs, GPUs, and FPGAs. Nevertheless, comparing with these conventional implementation approaches, that we consider as baseline, still provide useful information about the performance level of our proposed architecture.

On the other hand, it is worth noting that the proposed in-memory computing approach doesn't alter the accuracy of the BNN model with respect to the conventional computing approach. It only provides a different way to implement the same computing operations on the same parameters and same data.

Comparison results are summarized in Table II. As shown in this table, our proposed semi-parallel and parallel architectures exhibit the lowest energy consumption regardless of the used NVM technology (STT/SOT). The SOT-based architecture reveals 49% to 99% reduction in terms of energy compared to the other implementations. Although the parallel architecture shows better inference speed than the semi-parallel one ($6.5\times$), the latter requires less number of Near-Memory Units ($M\times$). The choice between these two architecture models could depend on the application requirements. Furthermore, our proposed architecture with SOT technology reaches 264 to 273 image/sec/Watt throughput efficiency, which outperforms all others including the CPU, GPU, and FPGA solutions.

It is worth noting that the semi-parallel and fully parallel architectures have the same energy consumption although they have different implementation complexity. This is due to the fact that they perform exactly the same operations, yet with different level of parallelism. A slight difference appears due to the static energy consumed by the additional NMUs in the

parallel architecture. However, this static energy is negligible with respect to the predominant dynamic energy consumed by the resistive switching behavior of the MTJ cells.

In terms of inference speed, the proposed parallel architecture with SOT technology achieves $1.24\times$, $1.35\times$ and $3\times$ speedup compared to Read-SOT [35], PXNOR-BNN [3] and Intel Xeon E5-2640 processor (CPU) approaches respectively thanks to the high level of parallelism offered using MOL-based in-memory computing. On the other hand, the GPU and FPGA implementations reveal higher inferencing speed ($6.3\times$ and $1.6\times$ respectively) compared to our SOT-based parallel architecture. However, this higher speed comes at the cost of $41\times$ and $3.3\times$ energy consumption respectively. In this regard, it is worth noting that our architecture design adopts the CMOS 65 nm technology node. If the design is scaled to more recent technologies (e.g. 28 nm), we expect even better results in terms of speed and energy consumption. At last, although the GPU and FPGA implementations present better results in terms of speed, still they are less convenient for embedded applications such as IoT devices where area and energy consumption are the most crucial.

C. Hardware complexity evaluation

The proposed architecture, which is presented in two models, is mainly composed of interconnected CMEMs, near memory units and a control unit. The storage space of a CMEM unit should be sufficient to accommodate the Ifmaps, the resulting Ofmap and the results of intermediate computations of the network layer. Each CMEM unit in the BinaryNet model should have a minimum of 36 KB of space, for a total of 4.5 MB for all CMEM units. It is worth noting that these units are based on the conventional 1T1M crossbar arrays, thus can benefit from the promising 3D stacking technology of NVMs [37] [38]. Accordingly, the CMEMs crossbars can be stacked at the top of each other, leading to a compact design. In fact, it has not been possible to evaluate the area of the proposed architecture due to the lack of a layout model for the adopted MTJ devices. However, in order to get an estimation of the complexity, we evaluated the total number of components in each CMEM unit, particularly in the peripheral and intermediate driver. Table III presents the number of utilized hardware resources for a 34-bit width as well as for a general d-bit width CMEM unit. Here, it's worth noting that the reported number of resources grows when expanding the width d of a CMEM unit. Increasing the depth (i.e., number of wordlines) has no additional cost. Table IV presents the number of components involved in the near memory unit. The NMU is mainly composed of k-bit adders, constant comparators and registers. In the presented case study, as we employed 128 CMEM banks, the parallel architecture integrates 128 NMUs whereas the semi-parallel architecture integrates only one NMU.

On the other hand, the functionality of the control unit has been emulated using a python-script code, which automatically generates the commands and the address signals to the interconnected CMEM units. It is considered that this functionality can be handled using a host processor, which is programmed according to the desired application (network model, dataset, parameters). The use of a host processor ensures programmability requirement.

VI. Conclusion

In this work, we presented a novel in-memory computing architecture targeting efficient implementation of BNN. The proposed architecture follows what we called SIMM parallelism model to execute instructions inside multiple computational memories. It employs the advanced quantization algorithm of BNN and the promising MOL-based in-memory computing technique which is well adapted for parallel bitwise operations. The complex multiply-and-accumulate operations are replaced by parallel row-wise XNOR and popcounts. Moreover, the addition and normalization stage is replaced by Majority-Bin stage which is achieved through the proposed in-memory bubble sorting technique of channels. An efficient data mapping methodology has been deployed. For simulation and evaluation purposes, a python based environment with appropriate library and commands has been developed emulating the functionality of the adopted MOL-based computational memory architecture. A study on the CIFAR-10 BNN model has demonstrated that a proposed parallel architecture, implemented in SOT-MTJ technology, has obtained a notable performance improvement compared to recent relevant state-of-the-art works. The results show $1.24\times$ to $3\times$ speedup compared with Read-SOT, PXNOR-BNN and Intel Xeon E5-2640 processor (CPU) implementations. Moreover, the proposed parallel architecture outperforms all other approaches in terms of power and energy consumption, where 49% to 99% reduction is achieved in terms of energy cost. Besides, it reaches 264 to 273 image/sec/Watt throughput efficiency, which is much higher than all others including the CPU, GPU, and FPGA solutions. Finally, the proposed architecture is scalable as it is able to handle larger network sizes, and can in addition be compatibly applied to other types of emerging memory technologies.

References

[1] S. Yu, S. Jia, and C. Xu, "Convolutional neural networks for hyperspectral image classification," *Neurocomputing*, vol. 219, 2017.

[2] C. Chen, A. Seff, A. Kornhauser *et al.*, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE international conference on computer vision*, 2015.

[3] L. Chang, X. Ma, Z. Wang *et al.*, "PXNOR-BNN: In/With spin-orbit torque MRAM preset-XNOR operation-based binary neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, 2019.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[5] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, 2019.

[6] H. Qin, R. Gong, X. Liu *et al.*, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, 2020.

[7] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[8] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh *et al.*, "Memory devices and applications for in-memory computing," *Nature nanotechnology*, vol. 15, no. 7, 2020.

[9] M. Courbariaux, I. Hubara, D. Soudry *et al.*, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[10] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," *arXiv preprint arXiv:1711.11294*, 2017.

TABLE II
PERFORMANCE COMPARED WITH STATE-OF-THE-ART DESIGNS FOR THE CIFAR-10 BNN MODEL

Item	CPU [34] Intel Xeon E5-2640	GPU [34] NVIDIA Tesla K40	FPGA [34] Zynq-7000 SoC	Read-SOT [35] (SOT)	PXNOR-BNN [3] (SOT)	This work (semi-parallel) (STT/SOT)	This work (parallel) (STT/SOT)
Technology node	32 nm	28 nm	28 nm	45 nm	28 nm	65 nm	65 nm
Clock frequency	2.5 GHz to 3 GHz	745 MHz to 875 MHz	143 MHz	500 MHz	1.4 GHz	555 MHz / 1 GHz	555 MHz / 1 GHz
Execution time (ms/image)	13.2	0.68	2.68	5.35	5.83	50 / 28.1	7.7 / 4.3
Power (W)	95	235	4.7	2.1	1.41	0.15 / 0.13	0.96 / 0.88
Energy (mJ/image)	1254	159.8	12.59	11.23	7.58	7.5 / 3.82	7.5 / 3.82
Throughput efficiency (image/sec/Watt)	0.7	6.2	79.4	89	131.8	133.3 / 273.7	135.2 / 264.2

TABLE III
NUMBER OF PERIPHERAL COMPONENTS PER CMEM UNIT

Type	# Components (d-bit width CMEM)	# Components (34-bit width CMEM)
2:1 Mux	3d	68
Inverters	8d+6	278
Transmission Gate	2d	68
Pass-MOSFET	6d+4	208
Ref-resistor	2d	68

TABLE IV
NUMBER OF COMPONENTS IN EACH NMU

Near memory unit	3-bit adder	3-bit comparator	3-bit register	1-bit full adder
d-bit width CMEM & k=3	$\lfloor d/3 \rfloor$	$\lfloor d/3 \rfloor$	$\lfloor d/3 \rfloor$	$\lfloor d/3 \rfloor$
34-bit width CMEM & k=3	11	11	11	11

$\lfloor . \rfloor$ is the Floor operator

[11] S. Hamdioui, S. Kvatinsky, G. Cauwenberghs *et al.*, "Memristor for computing: Myth or reality?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 722–731.

[12] C.-J. Jhang, C.-X. Xue, J.-M. Hung *et al.*, "Challenges and Trends of SRAM-Based Computing-In-Memory for AI Edge Devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 5, pp. 1773–1786, 2021.

[13] K. A. Ali, M. Rizk, A. Baghdadi *et al.*, "Memristive Computational Memory Using Memristor Overwrite Logic (MOL)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, 2020.

[14] N. Talati, S. Gupta, P. Mane *et al.*, "Logic design within memristive memories using Memristor-Aided loGIC (MAGIC)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, 2016.

[15] S. Kvatinsky, G. Satat, N. Wald *et al.*, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 10, pp. 2054–2066, 2014.

[16] S. Zhou, Y. Wu, Z. Ni *et al.*, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[17] U. Karn, "An intuitive explanation of convolutional neural networks," *The data science blog*, 2016.

[18] M. Rastegari, V. Ordonez, J. Redmon *et al.*, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016.

[19] C.-X. Xue, W.-H. Chen, J.-S. Liu *et al.*, "A 1Mb multibit ReRAM computing-in-memory macro with 14.6 ns parallel MAC computing time for CNN based AI edge processors," in *IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019.

[20] W.-H. Chen, W.-J. Lin, L.-Y. Lai *et al.*, "A 16Mb dual-mode ReRAM macro with sub-14ns computing-in-memory and memory functions enabled by self-write termination scheme," in *IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2017.

[21] M. Hu, H. Li, Y. Chen *et al.*, "Memristor crossbar-based neuromorphic computing system: A case study," *IEEE transactions on neural networks and learning systems*, vol. 25, no. 10, 2014.

[22] C.-X. Xue, T.-Y. Huang, J.-S. Liu *et al.*, "A 22nm 2Mb ReRAM Compute-in-Memory Macro with 121-28TOPS/W for Multibit MAC Computing for Tiny AI Edge Devices," in *IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2020.

[23] W.-H. Chen, C. Dou, K.-X. Li *et al.*, "CMOS-integrated memristive non-volatile computing-in-memory for AI edge processors," *Nature Electronics*, vol. 2, no. 9, 2019.

[24] S. Kvatinsky *et al.*, "MAGIC—Memristor-Aided Logic," *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[25] X. Fang and Y. Tang, "Circuit analysis of the memristive stateful implication gate," *Electronics Letters*, vol. 49, no. 20, pp. 1282–1283, 2013.

[26] S. Li, C. Xu, Q. Zou *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *53nd Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[27] D. B. Strukov, G. S. Snider, D. R. Stewart *et al.*, "The missing memristor found," *nature*, vol. 453, no. 7191, 2008.

[28] S. Ikeda, K. Miura, H. Yamamoto *et al.*, "A perpendicular-anisotropy CoFeB–MgO magnetic tunnel junction," *Nature materials*, vol. 9, no. 9, 2010.

[29] H.-S. P. Wong, S. Raoux, S. Kim *et al.*, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[30] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," *Technical Report, University of Toronto*, 2009.

[31] Y. Zhang, S. Lin, R. Wang *et al.*, "When sorting network meets parallel bitstreams: a fault-tolerant parallel ternary neural network accelerator based on stochastic computing," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1287–1290.

[32] M. R. Alam, M. H. Najafi, and N. TaheriNejad, "Sorting in memristive memory," *J. Emerg. Technol. Comput. Syst.*, jan 2022, just Accepted. [Online]. Available: https://doi.org/10.1145/3517181

[33] A. K. Prasad, M. Rezaalipour, M. Dehyadegari *et al.*, "Memristive data ranking," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 440–452.

[34] R. Zhao, W. Song, W. Zhang *et al.*, "Accelerating binarized convolutional neural networks with software-programmable FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[35] S. Angizi, Z. He, and D. Fan, "PIMA-logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation," in *Proceedings of the 55th Annual Design Automation Conference*, 2018.

[36] Z. Wang, H. Zhou, M. Wang *et al.*, "Proposal of toggle spin torques magnetic RAM for ultrafast computing," *IEEE Electron Device Letters*, vol. 40, no. 5, 2019.

[37] Y. Cao, G. Xing, H. Lin *et al.*, "Prospect of spin-orbitronic devices and their applications," *IScience*, p. 101614, 2020.

[38] Y. Huai, H. Yang, X. Hao *et al.*, "High density 3d cross-point stt-mram," in *2018 IEEE International Memory Workshop (IMW)*. IEEE, 2018, pp. 1–4.

Khaled Alhaj Ali received his master degree in Signal, Telecommunications, Image and speech (STIP) in 2016, from the Lebanese University, Beirut, Lebanon. He received his PhD degree in Electronics from IMT Atlantique, France in 2020. He is now a post-doctoral researcher at the Mathematical and Electrical Engineering department of IMT Atlantique. His current research interest is focused on the use of emerging non-volatile memory technologies for in-memory computing and neural networks.

Amer Baghdadi is a Professor at IMT Atlantique/Lab-STICC laboratory. He received his Engineering degree in 1998, Master of Science degree in the same year and PhD degree in 2002, all from Grenoble INP (Institut National Polytechnique), France. Furthermore, he received the accreditation to supervise research (HDR) in Sciences and Technologies of Information and Communication in 2012 from the University of Southern Brittany, France. Prof. Baghdadi leads the Algorithm-Architecture Interaction (2AI) team in the Mathematical and Electrical Engineering department of IMT Atlantique. He has recently contributed to FlexDEC-5G, H2020 METIS, FANTASTIC-5G, and EPIC research projects. His research activities are mainly focused on embedded system design for applications in digital communications and neural networks, with a special interest in flexible implementations, application-specific processor design, hardware accelerators, NoC and MPSoC architectures, energy-efficient designs, processing-in-memory and non-volatile memory-based design approaches.

Elsa Dupraz (member IEEE) was born in Paris, France. She received the Master degree in advanced systems of radio-communications from ENS Cachan and University Paris Sud, France, in 2010, and the Ph.D. degree in physics from University Paris-Sud, France, in 2013. From January 2014 to September 2015, she held a Postdoctoral position wih ETIS (ENSEA, University Cergy-Pointoise, CNRS, France) and ECE Department, University of Arizona, USA. Since October 2015, she has been an Assistant Professor with IMT Atlantique. Her research interests lie in the area of coding and information theory, with a special interest on distributed source coding, LDPC codes, and energy-efficient channel codes.

Mathieu Léonardon received the M.Sc. degree from Bordeaux INP, Bordeaux, France, in 2015. He received the Ph.D. in electronics engineering from Polytechnique Montreal, Canada, and from the University of Bordeaux, France, in 2018. Between 2018 and 2019 he held a lecturer position at Bordeaux INP. He is an Associate Professor at IMT Atlantique in Brest, France, since 2020. His research interest are on hardware and software implementations. These include two main fields of application: signal processing and in particular Error-Correcting Code decoders on the one hand, and Deep Neural Networks on the other hand.

Mostafa Rizk received his Maitrise degree in Electronics, M.Sc in Biomedical Physics, and M.Sc in Signal, Telecom, Image, and Speech from the Lebanese University in 2007, 2008 and 2010 respectively. He received his Ph.D. degree in Sciences and Technologies of Information and Communication from Telecom Bretagne, France in 2014 and a Ph.D degree in Electronics and Telecommunication from the Lebanese University in 2015. Dr. Rizk has been a post-doctoral researcher at UBS University and Lab-STICC laboratory CNRS, France. Dr. Rizk has been an associate professor at LIU, Lebanon and associate researcher at IMT-Atlantique, France. Currently, Dr. Rizk is a researcher at Lab-STICC laboratory - UMR 6285 CNRS, France. His general research interests include both algorithm development and corresponding hardware/software co-design for embedded architectures, circuits and systems; NoC design and new MPSoC architectures based on emerging computing paradigms using memories and memristive technologies; ICT of drone systems; embedded machine learning and embedded computer vision.

Jean-Philippe Diguet is a CNRS director of research at Lab-STICC, Lorient/Brest, France. He received the Ph.D. degree from Rennes University (France) in 1996. In 1997, he has been a visitor researcher at IMEC (Belgium). He has been an associate professor at UBS University (France) until 2002. In 2003, he co-funded the dixip company in the domain of wireless embedded systems. Since 2004 he is a CNRS researcher at Lab-STICC, where he has been heading the MOCS team until 2016. He has been a visitor researcher at the University of Queensland, Australia in 2010 and an invited Prof. at Tohoku University, Japan in Nov. 2014 and May 2019, and at Univ. of Sa̋o Paulo, Brazil, in Nov. 2016. His current work focuses on various aspects of embedded system design: Designs and Tools for NoC-based MPSoC architectures including memory-based computing, Self-adaptivity for uncertain environments as autonomous vehicles and Design of dedicated hardware accelerators.