Bit-Line Computing for CNN Accelerators Co-Design in Edge AI Inference

Marco Rios, Flavio Ponzina, Alexandre Levisse, Giovanni Ansaloni, David Atienza

Abstract—By supporting the access of multiple memory words at the same time, Bit-line Computing (BC) architectures allow the parallel execution of bit-wise operations in-memory. At the array periphery, arithmetic operations are then derived with little additional overhead. Such a paradigm opens novel opportunities for Artificial Intelligence (AI) at the edge, thanks to the massive parallelism inherent in memory arrays and the extreme energy efficiency of computing in-situ, hence avoiding data transfers. Previous works have shown that BC brings disruptive efficiency gains when targeting AI workloads, a key metric in the context of emerging edge AI scenarios. This manuscript builds on these findings by proposing an end-to-end framework that leverages BC-specific optimizations to enable high parallelism and aggressive compression of AI models. Our approach is supported by a novel hardware module performing real-time decoding, as well as new algorithms to enable BC-friendly model compression. Our hardware/software approach results in a 91% energy savings (for a 1% accuracy degradation constraint) regarding state-of-the-art BC computing approaches.

Index Terms—Edge Artificial Intelligence, In-Memory Computing, Hardware/Software Co-Design, Convolutional Neural Networks, Low-power Software Optimization.

1 Introduction

Thanks to their ability to extract abstract information from raw data acquisitions, Machine Learning (ML) algorithms such as Convolutional Neural Networks (CNNs) are fostering a revolution in multiple and diverse fields, ranging from personal mobility to health-care. Nevertheless, the increased accuracy of recent CNN models comes at the cost of massive memory requirements and intense workloads [1].

These downsides are particularly important for edge devices running artificial intelligence algorithms, a scenario named edge AI in literature [2]. Computational efficiency is key in edge AI because applications often have to abide to real-time constraints. Such constraints have to be met within tight computing and energy budgets, commonly orders-of-magnitude lower at the edge when compared to the cloud, thus requiring careful optimization of hardware and software. The main avenues towards the optimization of ML workloads leverage the high level of parallelism and robustness of these algorithms.

In the field of CNNs, parallelism is enabled by their structured and repetitive computing patterns, based on Multiply-ACcumulate (MAC) instructions, millions of which are employed to implement their convolutional and fully connected layers. Indeed, a high degree of data reuse is present both when convolving filters with activations (in convolutional layers) and when executing matrix-vector products (in fully connected ones). This characteristic can, as we do in this papers, be effectively harnessed by Single Instruction, Multiple Data (SIMD) computation strategies to increase efficiency and performance [3].

• *The authors are with the Embedded Systems Laboratory, École Polytechnique Fédérale de Lausanne (EPFL), Route Cantonale, 1015 Lausanne, Switzerland.*
E-mail: {marco.rios, flavio.ponzina, alexandre.levisse, giovanni.ansaloni, david.atienza}@epfl.ch.

Moreover, due to their robustness, CNNs can be optimized with very little, or no, accuracy drop, by either reducing the amount of required MAC operations or simplifying their computation. Quantization approaches advocate the use of fixed-point formats in contrast to floating-point, enabling the use of only few bits to represent the parameters (weights) and intermediate values (activations). Pruning strategies focus at a coarser granularity, seeking to skip weights and MAC computations that have little impact on the output quality. As detailed in Section 2, pruning and quantization are often combined in state-of-the-art edge AI strategies.

Besides software optimization, the rise of edge AI has motivated the computer architecture and hardware research community to introduce dedicated designs. Approaches range from processors-based solutions, such as the ultra-low-power PULP multi-core in Rossi *et al*. [4], to custom accelerators, e.g., Eyeriss in Chen *et al*. [3]. In this context, In-Memory Computing (IMC) architectures are particularly appealing, as computation inside memory avoids energy-expensive data movements in-between processing and storage components, while the parallelism made available by the regular structure of memory banks presents a good opportunity to support the SIMD patterns in CNNs.

A promising implementation of the IMC paradigm, that we focus on in this manuscript, is that of Bit-line Computing (BC) [5]–[9]. Based on conventional high-density SRAM, BC architectures can be seamless integrated in CMOS technology and target different levels in memory hierarchies. Moreover, BC supports a very high level of SIMD parallelism through the use of multiple subarrays at the same time. Finally, BC implementations require very little area overhead at the memory periphery to implement shift-add operations, which can then be chained to compute MACs. The above-mentioned approach is particularly beneficial when small-bitwidth operands are considered, as those require a reduced number of shift-adds.

This work addresses the fundamental hardware/software co-design challenge by providing a holistic framework for the optimization, deployment, and execution of CNN models on a BC architecture for edge computing. We combine a novel BC-aware CNN optimization strategy with a highly optimized BC architecture. Both support fine-grained quantization and pruning in fully connected and convolutional layers. Moreover, leveraging the statistical distribution of weight values in CNNs, our methodology features a novel weight encoding strategy both during CNN optimization and in the BC hardware implementation. The strategy, named Generic Convolutional Weights (GCW) encoding, uses few bits to encode weight values that appear more frequently, and a higher number of bits for those that are only rarely used, reducing model sizes by up to 4x in our experiments. A dedicated pipeline is in charge of decompressing the model representation at run time, without any impact on performance, converting it to a sequence of BC operations. Operations are then executed in parallel on multiple memory subarrays, greatly reducing run-time.

In summary, the contributions of this paper are:

- We present a synergic hardware and software framework that employs fine-grained bitwidths, data compression, and in-memory parallel computing to support edge AI applications with a very high degree of energy efficiency.
- We detail the characteristics of a novel BC circuit implementation, able to effectively execute parallel in-memory MAC operations among operands of varying bitwidth.
- We introduce a strategy to automatically map on BC subarrays the parameters of convolutional and fully connected layers, maximizing operations parallelism and data reuse.
- We present a CNN model compression, called GCW coding, to compress CNN models losslessly. Also, we describe a corresponding decoding circuit operating at run-time, with little area and no timing overhead.

The remainder of the paper is organized as follows. Section 2 discusses related works on CNN optimization/compression and on IMC architectures. Section 3 introduces an overall view of the proposed framework. Section 4 focuses on the software optimization by detailing the CNN optimization methodology and the compression approach. Next, Sections 5 and 6 present the proposed BC memory array architecture and the pipeline circuit for GCW decoding, respectively. Section 7 describes the mapping of convolutional and fully connected layers of CNNs on the BC architecture. Section 8 provides details on the experimental setup, while our achieved results are presented in Section 9. Finally, Section 10 concludes the paper.

2 Background

2.1 Convolutional Neural Networks

CNNs process input data employing a layer-based structure, where increasingly more abstract features are extracted in deeper layers. The compute-intense workload and large memory requirements of CNNs are mostly due to convolutional (CONV) and fully connected (FC) layers. For both layer types, the number of MAC operations is related to the size of input and output features, with several millions of MACs being common requirements in recent CNNs [10].

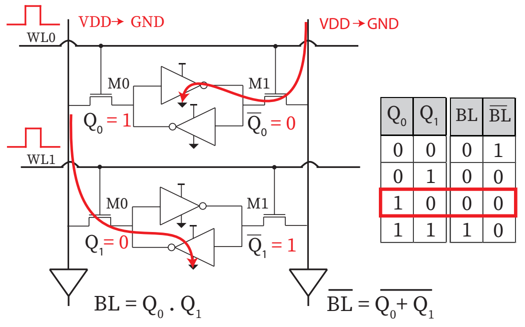
In CONV layers, three-dimensional matrices of CNN weights (named filters), are convolved over three-dimensional input feature maps, producing one output element for each filter position in the input. Conversely, FC layers (which are usually included after convolutional ones) compute linear transformations, multiplying the input feature vectors by the weight matrixes. FC and CONV layers have different data access patterns. In contrast to convolutions, weights in fully connected layers are used only once during an inference, because each column of the weight matrix multiplies the input vector to produce one output element. This characteristic is taken into account in the data mapping strategy discussed in Section 2.3. Recently, few other works have proposed methods to map large CNN layers in constrained IMC resources. As opposed to our focus on BC architectures, they consider crosspoint arrays [11], [12], and provide solutions only for convolutional layers [13].

2.2 CNN models compression

Pruning and quantization are the most common approaches exploiting the inherent redundancy in CNNs to reduce their complexity, hence supporting their deployment in constrained devices. In *pruning*, either individual weights [14] or entire convolutional filters [15] are removed, achieving model compression higher than 10x [16]. Instead, in *quantization* techniques, the weights and activations comprising CNNs models are represented using low-bitwidths integer data representations instead of floating-point numbers [17]. Quantized CNN models have a smaller memory footprint than floating-point ones. Furthermore, they use less complex integer hardware for computing MAC operations, which also reduces energy requirements. It has been shown that 8-bits quantization can be usually implemented without affecting the initial CNN accuracy [18].

A further approach to compress CNN models is that of weight *encoding*. It is often applied after quantization, as low-bitwidth representations constrain the set of admissible values. Encoding can be implemented employing different strategies. Codebook-based strategies limit the number of unique weights and store them in small look-up tables (i.e., named codebooks), encoding the baseline model into a set of binary indexes that address specific code-words [19]. Another weight encoding approach is to leverage the statistical distribution of weight values to compress their representation, employing shorter code-words for the most used values and longer code-words for rarer ones [20], [21].

Since encoding weight values does not change CNN models, but only operates on data representation, it does not degrade accuracy. On the other side of the coin, run-time decoding may introduce overheads in time, area, and energy requirements. The authors of [20] use Huffman codes to index a codebook storing a constrained set of CNN weight values. While our GCW strategy (detailed in Section 4) has some similarities with respect to Huffman coding, it does



Q_0	Q_1	BL	$\overline{BL}$
0	0	0	1
0	1	0	0
1	0	0	0
1	1	1	0

Figure 1. Bit-line computing concept. Two word-lines are activated in the same clock cycle. The discharge of the bit-lines result in the bit-wise operations AND and NOR between the accessed words.

not require explicit look-up tables, minimizing the cost of its hardware implementation.

2.3 IMC architectures

In-Memory Computing (IMC) relies on smart memories to perform computing tasks in a highly parallel way. IMC is especially attractive for applications that are computationally and memory-intensive, while presenting regular patterns and few (or no) data-dependent branches or loop-carried dependencies. CNNs do present these characteristics, and are therefore well-suited for IMC execution.

IMC architectures can be broadly divided into two categories. Crosspoint architectures employ meshes of variable resistances to encode weights and arrays of analog-to-digital converters to read-back results [22]. They can efficiently perform matrix-vector multiplications, but are prone to noise due to manufacturing variability, temperature, and voltage fluctuations [23], [24]. Moreover, they require the challenging co-integration of digital, analog, and non-volatile technologies.

A second strategy, that we focus on in this paper, is that of Bit-line Computing (BC) [25]. This approach can be readily integrated into existing SRAM memories and hierarchies by adding the capability to concurrently activate multiple memory words and employing sense amplifiers to perform bit-wise operations. As the name suggests, bit-line Computing relies on the behavior of the bit-lines (BL and $\overline{BL}$) when two Word-Lines (WLs) are activated in the same cycle clock.

Such a behavior is exemplified in Figure 1, which shows two SRAM bit-cells (each realized as two cross-coupled inverters) connected to the same BLs. When the two WLs of the cells are activated, if either one or both of the cells store the value ’0’, the voltage on the BL wire discharges to the ground through one or both of the cells access transistor (M0) and the NMOS of their inverter. Notice that the only case in which BL remains at Vdd is when both cells store the value ’1’. Thus, the BL connection behaves as the logic AND gate. Dually, the negated BL signal ($\overline{BL}$) only retains a high voltage if both cells store the value ’0’ (hence $\overline{Q_0}$ and $\overline{Q_1}$ are both ’1’), implementing the functionality of a NOR gate.

This setup can induce data-corruption due to undesirable currents flowing between the bit-cells. In order to address this issue, [25] reduces the operating frequency and [26] adopts 10T bit-cells. However, these solutions either

Table 1. Assignment of operands for fully connected and convolutional layers

Layer type / Operand	**CONV**	**FC**
Multiplicand In-Memory Operand (IMO)	Activations	Weights
Multiplier Broadcasted Operand (BO)	Weights	Activations

decrease the performance or imply large area overheads. Recent works [5], [6] have shown that such operations can be reliably performed at high clock frequencies if the activated WLs belong to different subdivisions of SRAM arrays, termed Local Groups (LGs). Only memory cells in the same LG share short-distance vertical connections (local bit-lines, LBLs). During a read access, cell values are propagated via LBLs to the LG Periphery (LGP) circuit which features sense amplifiers and read/write ports. In-memory bit-line operations are then performed after those sense amplifiers, effectively protecting memory cells from data corruption.

Further logic is then employed at the array periphery to derive additions from bit-wise operations. Finally, multiplication instructions are performed as a sequence of shift-adds between two memory words storing a) one of the two multiplication operands and b) the accumulated partial results (details are provided in Section 5.3).

This strategy differentiates between the two inputs of a multiplication, as only one of them (*In-Memory Operand*, IMO) resides in the memory array, while the other (*Broadcasted Operand*, BO) is streamed in the memory one bit at a time. Indeed, a single BO can be streamed to multiple memory subarrays at once to perform multiplications in parallel sharing one operand (hence its name). As shown in Table 1, due to the differences in access patterns and to maximize data reuse, in CONV layers we consider weights as BOs and activations as IMOs, while for FC layers we operate the opposite choice.

Although shift-adds require several cycles to execute multiply or MAC instructions, high performance can be achieved in BC arrays by (i) word-level parallelism inside a single subarray, (ii) partitioning the SRAM into subarrays to perform parallel operations, (iii) workload optimization to reduce the bitwidth of streamed BO operands, reducing the cycle-count of multiplications. The effectiveness of (i) and (iii) is highly influenced by quantization and pruning, as they determine the number of shift-adds required by a multiplication. Nonetheless, current works do not explore this interdependence in detail. As an example, [17] adopts a fixed 16-bits representation for activations and an 8-bits one for weights. We instead show that important efficiency gains can be harnessed when employing more aggressive stances such as heterogeneous per-layer schemes, and that these can be supported with little hardware overhead.

3 Framework Overview

Figure 2 provides a bird’s eye view of our framework. Starting from a (floating-point, non-optimized) description of a CNN the framework provides a pathway to co-optimize the CNN implementation and the BC computing array executing it. It also optimizes the mapping of software onto hardware.

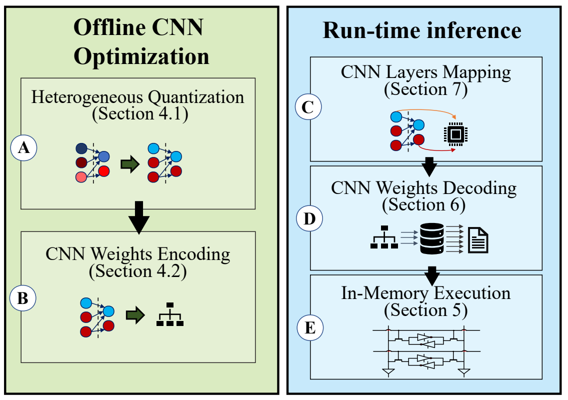
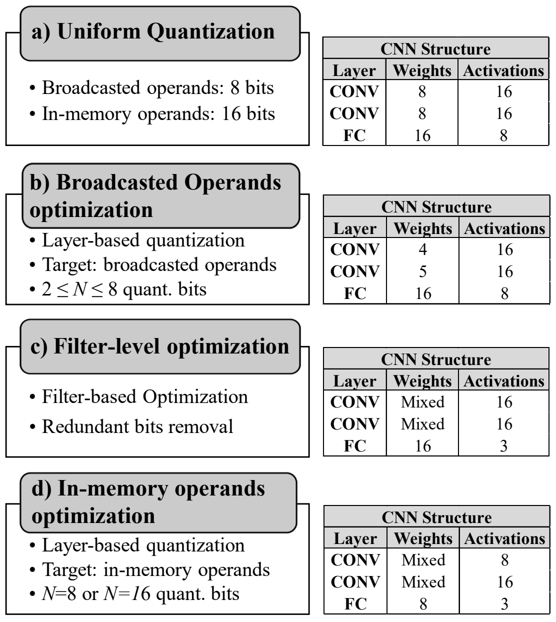

Offline CNN Optimization
Heterogeneous Quantization (Section 4.1)
A
CNN Weights Encoding (Section 4.2)
B
Run-time inference
CNN Layers Mapping (Section 7)
C
CNN Weights Decoding (Section 6)
D
In-Memory Execution (Section 5)
E

Figure 2. Overall view of our HW-SW co-design framework, showing algorithmic optimizations (left), mapping, and execution on BC hardware (right).



a) Uniform Quantization

- Broadcasted operands: 8 bits
- In-memory operands: 16 bits

CNN Structure		
Layer	Weights	Activations
CONV	8	16
CONV	8	16
FC	16	8

b) Broadcasted Operands optimization

- Layer-based quantization
- Target: broadcasted operands
- $2 \leq N \leq 8$ quant. bits

CNN Structure		
Layer	Weights	Activations
CONV	4	16
CONV	5	16
FC	16	8

c) Filter-level optimization

- Filter-based Optimization
- Redundant bits removal

CNN Structure		
Layer	Weights	Activations
CONV	Mixed	16
CONV	Mixed	16
FC	16	3

d) In-memory operands optimization

- Layer-based quantization
- Target: in-memory operands
- N=8 or N=16 quant. bits

CNN Structure		
Layer	Weights	Activations
CONV	Mixed	8
CONV	Mixed	16
FC	8	3

Figure 3. Workload-aware quantization and pruning methodology (left). Running example (right).

Application-level optimizations, detailed in Section 4, combine non-uniform quantization schemes and encoding methods to obtain an optimized model that can be efficiently executed in-memory. In more detail, we first include a resource-aware CNN quantization strategy that reduces workload and memory requirements (Figure 2.A). This stage consists in an iterative approach, where the bitwidth of weights and activations in convolutional and fully connected layers is optimized while abiding by an accuracy threshold. Then, due to the characteristic distribution of CNN weights, GCW encoding further compresses the quantized CNNs (Figure 2.B). GCW uses smaller bitwidths for the weight values appearing more frequently and larger bitwidths for those appearing only sporadically.

Because of the typically large size of intermediate features in CNNs, we decompose convolutions and the matrix-vector operations in FC layers into smaller computing blocks (Figure 2.C). This tilling process, focus of Section 7, allows large CNN models to be accelerated in limited-sized memories, minimizing the number of data transfers while exploiting parallelism to increase performance.

When processing convolutional layers, weights are decoded at run-time during execution, with minimal overhead (Figure 2.D). We describe the circuit performing weights decoding in Section 6. The decoder first translates parameters to their two's complement representations, and then converts them to a sequence of BC instructions, which govern the execution of our memory arrays (Figure 2.E). The design of the BC arrays and their features, including support for heterogeneous quantization, are detailed in Section 5.

4 Algorithmic-level CNN optimization

Algorithmic optimizations (Figure 2.A) aim at decreasing the workload and memory requirements of a CNN model with BC-aware transformations. The optimization process consists of two stages: first, a heterogeneous quantization and pruning step reduce the bitwidth and the number of weights and activations in convolutional and fully connected layers. Next, the quantized weights are encoded using variable-bitwidth codes, further reducing memory requirements.

4.1 Heterogeneous Quantization

Per-layer quantization enables aggressive model compressions. Indeed, layers in CNN may (and, usually, do) have different degrees of robustness, with quantization-sensitive layers requiring larger bitwidth for activations and parameters. Heterogeneous schemes have therefore the potential to reach better trade-offs between accuracy and model size. However, they also expose a much larger design space than uniform alternatives.

To navigate it, we introduce an iterative process that reduces the size of in-memory and broadcasted operands (IMOs and BOs, as defined in Table 1), according to the scheme in Figure 3. As a running example, the figure considers in its rightmost part a running example referring to a CNN with two convolutional and one fully connected layers, which we will follow in the rest of this section.

The input models in our optimization flow are homogeneously quantized CNNs, employing 16-bits IMOs and 8-bits BOs (Figure 3(a)). As shown in [17], CNNs at these quantization levels have indistinguishable accuracies with respect to floating-point implementations. In the first optimization step (Figure 3(b)), the bitwidth of the BOs is independently tailored for each layer. To this end, we attempt to reduce bitwidths starting from the layer having the highest number of MAC operations (and, therefore, the highest potential for savings). Then, we retrain the network for a small number of epochs and check the obtained accuracy of the new configuration. If the accuracy degradation exceeds a user-defined threshold, we backtrack (we considered 1% and 5% maximum degradations for the experiments in Section 9) . In a similar fashion, we then iteratively target the layers having the second, third, etc. most numerous MAC operations. Once all layers have been processed once, we further try to reduce the bitwidth of the BOs in the highest-workload layer from which we haven't previously backtracked. The iteration continues until no further bitwidth reductions are possible in the BOs of any layer.

This phase is followed by a filter-level optimization,

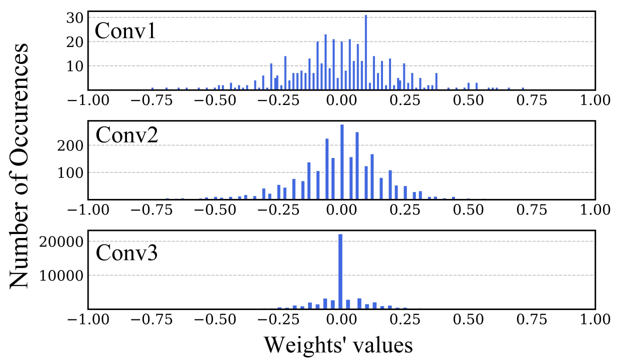


Figure 4. Weights distribution in the three convolutional layers of LeNet-5.

which focuses on convolutional layers only. We observe that a large amount of CNN filters do not use the entire value range when representing weights, especially after the above-mentioned aggressive quantization. As illustrated in Figure 3(c), we hence drop, without loss of accuracy, the most significant bits (MSbs) on a per-filter base if allowed by weight ranges, correspondingly scaling convolution outputs. For example, if the value range of the BOs in a filter is $\subset [-0.25, 0.25)$, 2 MSbs can be dropped, and outputs should be divided by $2^{Dropped\_bits} = 4$. Additionally, in this stage, filters having all weights equal to zero are entirely deleted.

The last step of the optimization flow (Figure 3(d)) performs the tailoring of the IMOs bitwidths. It leverages the word-level parallelism supported by the BC arrays (as described in Section 5.4). Similarly to the approach followed for BOs, we attempt to reduce the bitwidth of IMOs on a per-layer basis. However, quantization steps are coarser in this case, as they must abide to the sub-word formats supported in hardware. In our experiments, we admit 1x16-bit and 2x8bit sub-words, with the latter resulting in 2x reduction in execution time[1].

4.2 Generic Convolutional Weights Encoding

GCW compression further reduces the memory required to represent parameters in convolutional layers. Our strategy leads to an efficient implementation of run-time decoding, which we describe in Section 6. It takes advantage of the limited set of values that can be assumed by quantized weights, as well as their characteristic statistical distribution.

We observe that weights in quantized CONV layers predominantly assume the value "0", even after removing filters only containing zero values. This trend is illustrated in Figure 4, which provides as examples the weights distribution in the three convolutional layers of LeNet-5 [2].

Additionally, narrow distributions centered around zero indicate that small (quantized) weight values appear much more frequently than larger-magnitude ones. These findings open the path for an encoding scheme that, similarly to Huffman coding, employs code-words of variable length. Highly occurring values are coded using low-bitwidth representations, while less frequent values are mapped to higher bitwidth codes. Given the distributions of CNN weight values, this choice translates into employing the minimal bitwidth (1 bit) for the value "0", small bitwidths for the values close to zero and large bitwidths for values of high magnitude.

Decimal	GCW Coding	
Value (A)	Code	Size
$\frac{[-9, 2^{(N-1)}]}{2^{(N-1)}}$	10000 & bin $_N$(A)	N+5
$\frac{[-1, -8]}{2^{(N-1)}}$	1 & bin $_4$ (A)	5
0	0	1
$\frac{[1, 7]}{2^{(N-1)}}$	1 & bin $_4$(A)	5
$\frac{[8, 2^{(N-1)}-1]}{2^{(N-1)}}$	10000 & bin $_N$ (A)	N+5

Figure 5. Generic Convolutional Weights coding scheme, where N indicates the quantization level values before coding. The code uses fewer bits to represent the most frequent symbols.

The GCW encoding scheme is illustrated in Figure 5, with N being the number of quantized bits. Weights are assumed to be normalized in the range $[-1, 1)$. They are divided into five intervals, symmetric with respect to zero (as shown in the leftmost column). Values close to zero (other than zero itself) are represented with 5-bit code-words. A 1-bit prefix is appended to the two's complement 4-bit representation of the corresponding value.

Other, seldomly used, code-words are derived by appending a fixed 5-bit prefix to their N-bit representation.

For $N = 8$, 13 bits are required to represent large-magnitude values, but only 5 bits for low-magnitude ones (and only 1 bit for the value '0'). Note that, for $N < 5$, long code-words are never generated and the coding only generates different code-words lengths for zero and non-zero weights.

5 Bit-line Computing Architecture

5.1 BC array organization

As commonly done in standard SRAMs, the BC array architecture is divided into several subarrays. Each subarray is capable of performing a MAC operation between a broadcasted operand (which must be the same for all subarrays) and an in-memory operand, local to each subarray. Figure 6(a) shows the subarray organization. The Word-Lines (WLs) are evenly separated into Local Groups (LGs), which contain the memory cells and the LG Periphery (LGP). The role of the LGP is to electrically isolate SRAM cells from each other, allowing in-memory operations among words in different local groups at high speed without data corruption. LGPs logic also performs bit-wise negation and shift operations, as detailed in the following. At the bottom of the subarray, the Bit-line Computing Unit

1. While in principle our approach could be extended to 4 bits or 2 bits per word, such settings would incur in unacceptably large accuracy degradations.

2. We obtained similar results for the other benchmark CNNs in Section 9.

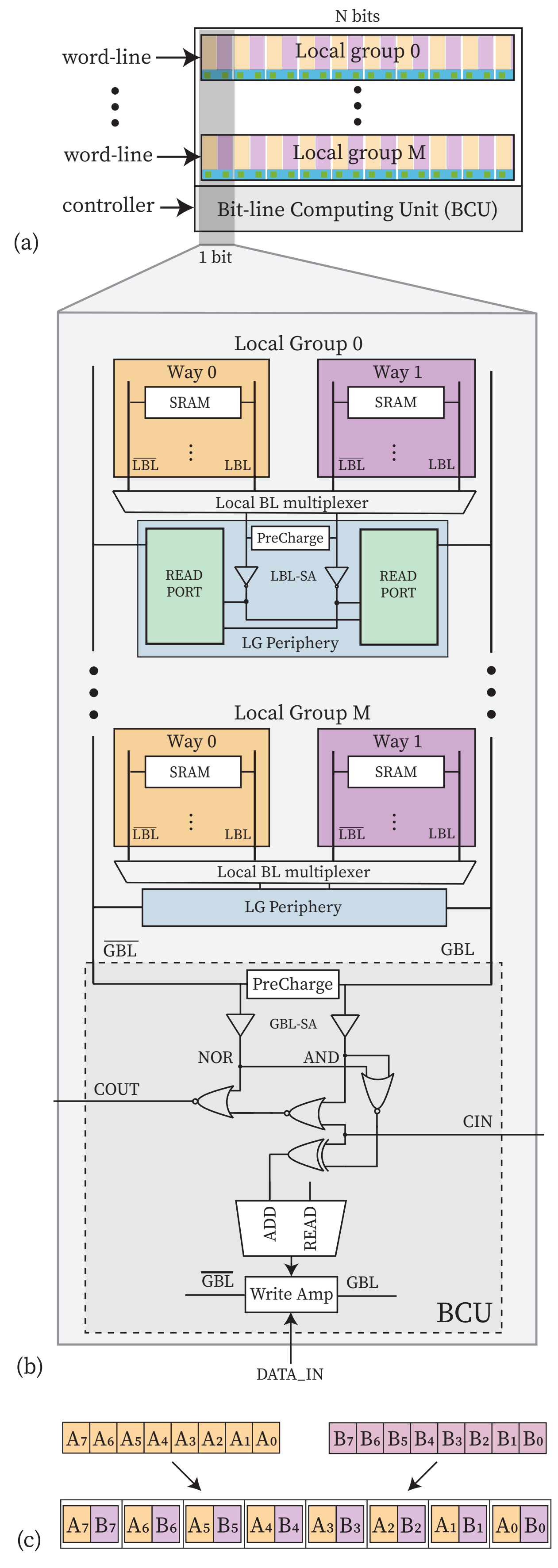


Figure 6. (a) Bit-line Computing subarray. (b) Highlight of one bit-column depicting the Local Group circuitry and Bit-line Computing Unit. (c) 8-bits two-ways interleaving.

(BCU) ripples a carry signal among bit columns, allowing the implementation of additions. Subarray operations are governed by a small controller, global to the entire array, which decodes BC operations and properly activate WLs.

Figure 6(b) highlights a single bit-column of the subarray, in which bit-cells are organized in Local Groups (LGs). Moreover, words in a LG are arranged with way interleaving (i.e., the same bit of multiple words are placed next to each other)[3]. Bit-cells belonging to the same local group and the same way share, via access transistors, share the same Local Bit Lines (LBLs). LBLs in a LG are connected to a multiplexer that acts as a way selector, allowing to connect LBLs to the sense amplifiers of the LG periphery and, through its read ports, to global bit-lines (GBLs). Finally, GBLs interface with the BCU, which, besides standard SRAM read and write ports, implements a 1-bit adder, rippling the carry in and out from/to neighboring bit-columns.

Our design based on way interleaving allows the memory array to be designed with high-density push-rule 6T bit-cells, while still maintaining enough clearance for the design of the LGP and BCU blocks, which must be vertically aligned with storage cells.

5.2 In-Memory Operations

The design illustrated above supports both standard read-write and in-memory operations, as detailed next.

5.2.1 *Write*

Write operations are performed by setting up the data on the GBLs through the write amplifier, which chooses data from two sources: the external input (DATA_IN in Figure 6) or the result of a performed BC operation. Thereon, the LGP write port [4] transfers the data from the GBLs to the LBLs. Finally, the WL is activated, switching on the access transistor and allowing the data to be written into the target SRAM bit-cell.

5.2.2 *Read*

Both LBLs and GBLs are pre-charged to `Vdd` before reading a word from the SRAM memory. Then, the WL is activated, switching on the access transistor and allowing one of the LBLs to discharge to the ground. Conversely, the other holds its charge, depending on the value stored in the bit-cell. The LBL Sense Amplifier (LBL-SA) consequently asserts a logic value and this signal is connected to the read port. GBLs assume the voltage value of the LBLs connected to the read ports, and the GBL Sense Amplifier (GBL-SA) asserts such value and outputs the read word.

5.2.3 *BC Instructions*

BC instructions always assume the format of `add(OP1, OP2)`, with the operands being allowed to be individually shifted and/or negated. OP2 can be set to zero if only shift/negation of one operand is required. The operation outcome can be either output from the memory or written back, which requires an additional cycle. `shift` and `negation` operations are performed on each operand in the LGP by the read ports, allowing the operands to be individually shifted/negated before the addition is performed.

3. For clarity, in Figure 6(b) we show an example with only two ways
4. Not depicted in Figure 6

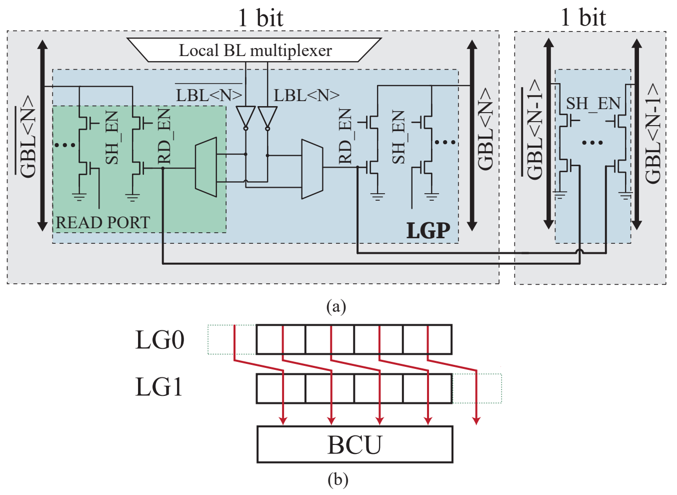


Figure 7. (a) Local Group Periphery circuit with extended read port, enabling shift and negation. (b) Data path of embedded shift.

Figure 7(a) presents a schematic of the read port circuitry. It offers two paths to discharge the GBLs to the ground, where each path is composed of two NMOS in series. Conventional read operation uses the path controlled by the Read Enable (RD_EN) signal and the LBL-SA output of the same bit. Shifts instead rely on the path controlled by the Shift Enable (SH_EN) signal and the LBL-SA output of a *neighbour* bit. This operation is defined as Embedded Shift (ES), since it is performed inside the LGP. Indeed, single-cycle, multiple-shifts operations can be supported in read ports by adding additional discharge paths from LBL-SA of farther bit columns, accomplishing design with a varying Number of Embedded Shifts (NES).

The LGP also embeds 2-to-1 multiplexers connected to both outputs of the LBL-SA. This design allows bit-wise negations to be supported in the LGP, as for this operation the multiplexers invert the LBL and $\overline{\text{LBL}}$. At the subarray level, values can consequently be arithmetically negated (i.e., in two's complement representation), by performing bit-wise negation in the LGP and asserting the carry-in of the least significant bit in the BCU.

5.3 Multiplications

5.3.1 Multiplication among binary numbers with partial products

In general terms, multiplications of two operands can be decomposed into partial products, which are summed up to retrieve the full product. Each partial product is found by multiplying each digit of the multiplier with the multiplicand and shifting the result to the left based on the position of this digit. In binary systems, the digits of the multiplier are either zero or one, thus, the partial products are either zero or the left-shifted multiplicand.

Figure 8 illustrates an example that shows a binary multiplication of two input operands, the IMO ($00100110_2 = 38_{10}$) and the BO ($10011_2 = 19_{10}$). Figure 8 also shows the required operations, where ACC is the partial products at each iteration. It can be noticed that operations are only performed with the IMO and product vector ACC,

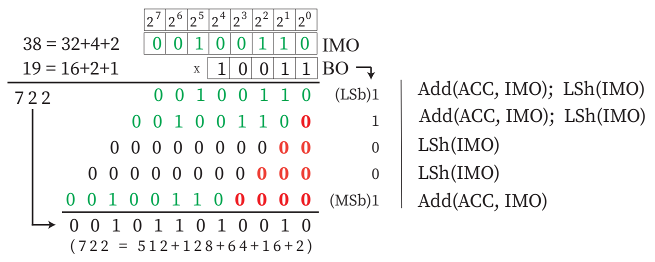

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
38 = 32+4+2   0 0 1 0 0 1 1 0   IMO
19 = 16+2+1   x 1 0 0 1 1   BO
722
0 0 1 0 0 1 1 0   (LSb)1   Add(ACC, IMO); LSh(IMO)
0 0 1 0 0 1 1 0 0   1   Add(ACC, IMO); LSh(IMO)
0 0 0 0 0 0 0 0 0 0   0   LSh(IMO)
0 0 0 0 0 0 0 0 0 0 0   0   LSh(IMO)
0 0 1 0 0 1 1 0 0 0 0 0   (MSb)1   Add(ACC, IMO)
0 0 1 0 1 1 0 1 0 0 1 0
(722 = 512+128+64+16+2)

Figure 8. Binary integer multiplication example, between the IMO ($00100110_2 = 38_{10}$) and the BO ($10011_2 = 19_{10}$), the multiplication is decomposed in shift-add instructions. LSh operations are left-shifts.

while each bit of the BO dictates which operation should be done at each step.

5.3.2 Multiplication in the BC array

The approach to multiplication illustrated above can not be executed as-is in the BC architecture. First, in Figure 8, the product ($1011010010_2 = 722_{10}$) requires more bits (10) compared to both of the operands (5 and 8-bits), which would result in an overflow in our BC scenario. Second, no support is provided for representing negative numbers, common in AI applications.

These two issues are addressed by supporting MAC operations in fixed-point (instead of integer) format, among signed numbers encoded in two's complement. We represent values using 1 bit for the integer part, and n bits for the fractional part (a format commonly indicated as $Q1.n$). Thus, the values are always in the range $[-1, 1)$. Since two's complement is used, the most significant bit is '1' for negative numbers and '0' for positive ones.

Multiplication on the BC architecture can then be performed by employing sequences composed by the following two instructions:

(i) Addition of two operands, both arithmetically right-shifted (i.e., with sign extension) by one bit. One operand is the partial product ACC, while the second is either 0 or the IMO, depending on the BO bit.
(ii) Addition between two operands, where one operand is the partial product ACC and the other is either 0 or the 2's complement of the IMO.

Instruction (i) is performed at every iteration step from bit 0 to bit $N-1$ of the BO, while operation (ii) is used for bit N of BO (its most significant bit).

Figure 9(a) shows an example of a multiplication among two two's complement numbers in the Q1.7 and Q1.4 formats, with the result in Q1.7. Binary digits are the same as the ones depicted in Figure 8, however, in Figure 9 the associated values are now IMO = $00100110_2 = 0.296875_{10}$ and BO = $10011_2 = \text{-}0.8125_{10}$.

Figure 9(b) shows the operations required to perform the multiplication, in the case of only supporting shifts of 1 bit (NES = 1). Notice that the bitwidth of the partial products does not increase at each iteration, and no overflow occurs. Instead, truncations induce small errors in the computed product (0.4% in the example).

Crucially, all the listed iterations include operations which are supported in the BC array, and each iteration can be executed in a single clock cycle. As presented in Figure 7,

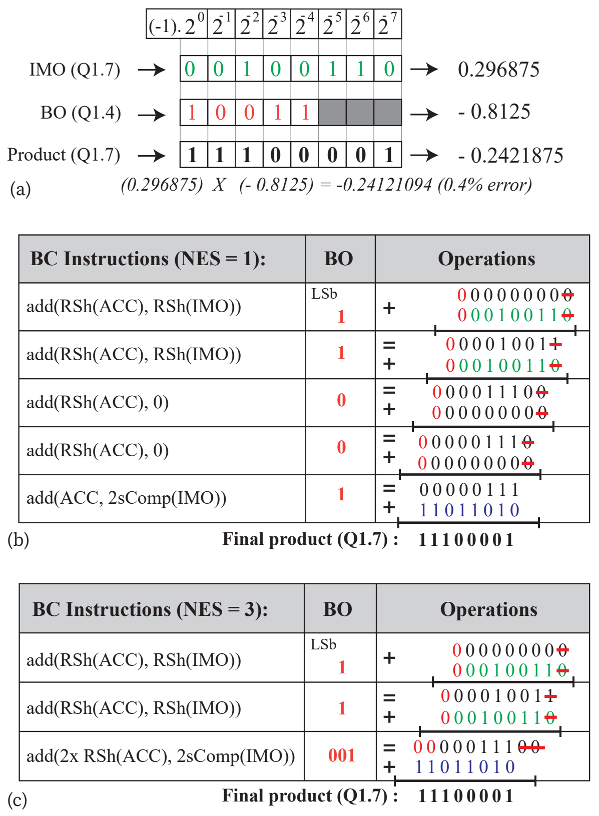


Figure 9. (a) Two's complement fixed-point multiplication example between the IMO expressed in Q1.7 and the BO expressed in Q1.4. The BC instructions for (b) NES = 1 and (c) NES = 3. RSh operations are right-shifts.

right-shifts are provided by read ports in Local Groups, and can execute concurrently with additions, since the BCU computes additions at the subarray periphery. Similarly, the addition of the partial product with the two's complement of the multiplicand, also requires one clock cycle as the bit-wise negation is done in the LG periphery, while the addition and the assertion of the carry-in is executed in the BCU.

5.3.3 *Using Multiple Embedded Shifts*

The support for multiple embedded shifts (NES>1) can effectively speed-up computation, at the cost of added complexity in the read port. Indeed, in the case of NES = 2, 2 bits of the BO can be processed in one iteration, provided that the first one is equal to zero ("00" and "01"), since in this case only one addition, or none, is required after two shifts. For NES = 3, BO sequences of bits with two leading zeroes ("000" and "001") can be processed in a single clock cycle. Figure 9(c) shows the BC instructions to execute the same multiplication as Figure 9(b), but on a BC array with NES = 3. In this configuration, only 3 operations are required (instead of 5 when NES = 1), resulting in a speed-up of 1.67x. A large number of supported NES can provide diminishing benefits, as long sequences of zeroes in the BO can be rare. Overly large NES values may also be detrimental, as they impact the complexity of LGP read ports [7].

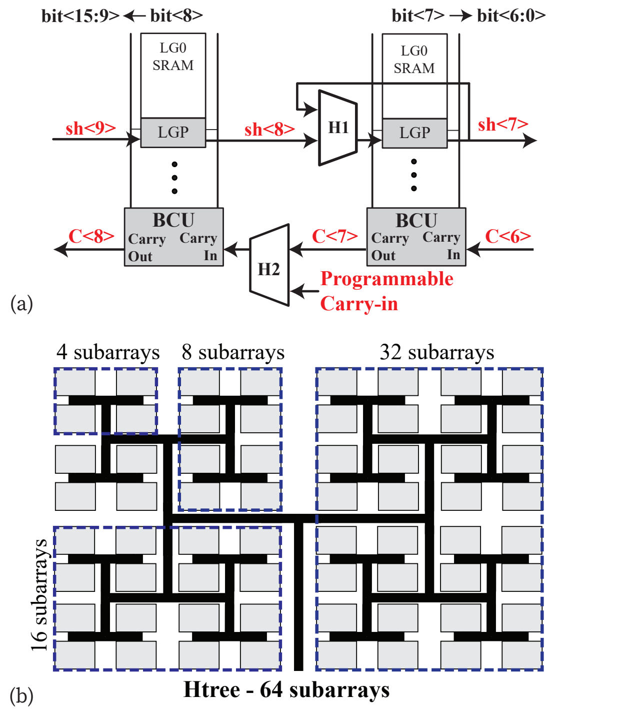


Figure 10. (a) Word-level parallelism support for 1x16bit and 2x8bit modes, considering NES = 1. (b) H-tree organization.

5.4 In-Memory Parallelism

5.4.1 *Word-level Parallelism*

The BC architecture allows to store in each memory location either a single 16-bit word (in Q1.15) or two 8-bit ones (each in Q1.7) [5]. In-memory instructions can hence be executed on 1x16bit or 2x8bit word formats, enabling in the latter case two multiplications simultaneously in a single subarray.

To implement word-level parallelism, connections between the 7^{th} and 8^{th} bit-column are configurable. In particular, right shift operations on the 7^{th} bit-column can either be connected to the 8^{th} bit-column (in 1x16bit mode), or to itself (to implement sign extension in 2x8bit mode). Similarly, the carry-in of the 8^{th} bit-column can be either connected to the 7^{th} bit-column (1x16bit mode), or dictated by the performed operation to implement two's complement (2x8bit mode).

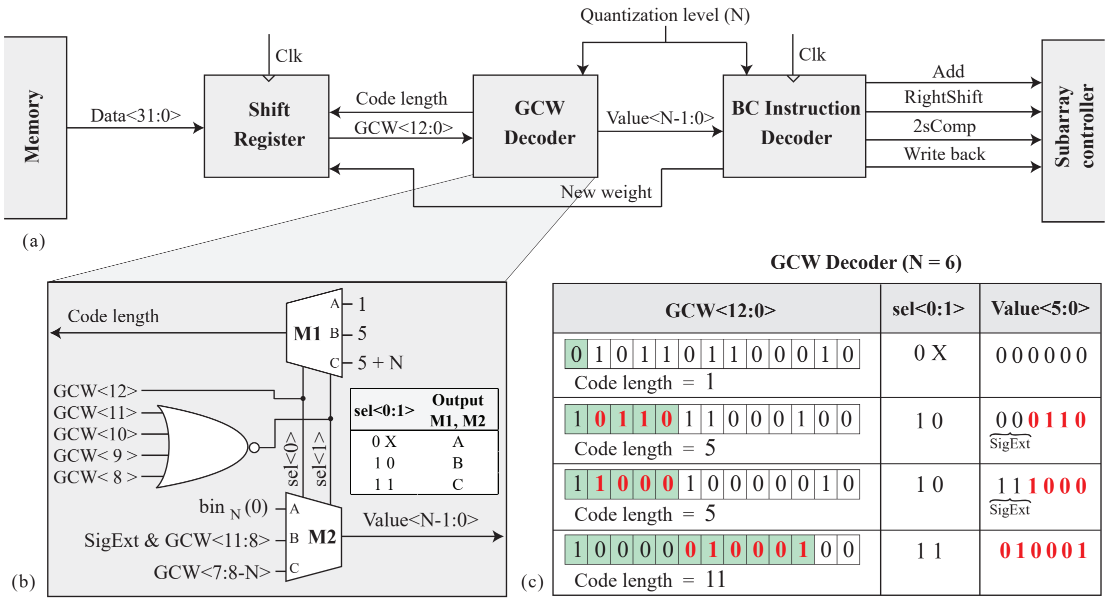
Hence, two additional multiplexers are required (H1 and H2 in Figure 10). The first one, at LG peripheries, connects or disconnects the shift-right signal from the 8^{th} to the 7^{th} bit-column, while the second, in BCUs, operates similarly for the carry signal between the 7^{th} and the 8^{th} bit columns.

5.4.2 *Array-level Parallelism*

Multiple subarrays are connected in an H-tree configuration, as shown in Figure 10(b). This organization ensures that all the signals transmitted from/to the array periphery have the same distance to all the subarrays, equalizing the delays and minimizing critical paths.

The H-tree interconnect is active before and after computation to transfer IMO inputs to subarrays, and retrieve results. During computation, the H-tree broadcasts one BC in-

5. Word-level parallelism does not impact normal (non-BC) reads and writes.



GCW<12:0>	sel<0:1>	Value<5:0>
0 1 0 1 1 0 1 1 0 0 0 1 0 Code length = 1	0 X	0 0 0 0 0 0
1 0 1 1 0 1 1 0 0 0 1 0 0 Code length = 5	1 0	$\underbrace{0\,0}_{\text{SigExt}}$ 0 1 1 0
1 1 0 0 0 1 0 0 0 0 0 1 0 Code length = 5	1 0	$\underbrace{1\,1}_{\text{SigExt}}$ 1 0 0 0
1 0 0 0 0 0 1 0 0 0 1 0 0 Code length = 11	1 1	0 1 0 0 0 1

Figure 11. (a) Block diagram of the pipeline circuit to extract BC operations from the compressed CNN model. (b) GCW decoder circuit-level design (c) GCW decoding example with N = 6.

struction related to BO bits to all subarrays at each clock cycle, exploiting the reuse of BOs in multiple MAC operations to achieve a high degree of parallelism. Importantly, this approach allows a fine-grained flexibility in the bitwidth of BOs, which may assume arbitrary values, leading to a fine-grained control of trade-offs between model cost (size, energy, time-per-inference) and accuracy.

6 Real-time Data Decompression

To reduce the size of CNN models, hence memory requirements, convolutional weights are stored in an encoded form, according to the Generic Convolutional Weights (GCW) representation introduced in Section 4.2. At run-time, a pipeline circuit decodes them, deriving the corresponding BC instructions. Figure 11(a) depicts the pipeline stages of the decoder. First, a shift register reads a memory word containing multiple GCW-encoded weights, possibly of different bitwidths. Then, the GCW decoder decompresses weights into their Q1.n representation. Finally, the BC instruction decoder converts these values into a set of BC instructions, which are broadcasted to the subarrays.

6.1 Shift Register

For each convolutional filter, GCW-encoded weights form a bit-stream where each value is represented with 1 bit (for the "0" value), 5 bits (for values close to 0), or up to 13 bits (otherwise). Hence, the shift register, which is filled by 32-bits memory words at a time, usually holds multiple GCW code-words. When the decoder requires a new weight, it examines the first 13 bits of the shift register (`GCW<12:0>` in Figure 11(a)), determining which bit-field blue(first bit, first 5 bits, or first 5+N bits, where N is the quantization level) contains the next code-word and advances the shift register according to the code length. The shift register is re-filled when less than 13 bits remain in its buffer, concatenating memory words. Code-words can hence cross the boundary of two subsequent memory words, preventing memory under-utilization.

6.2 GCW Decoder

The GCW decoder decodes the weights values according to their quantization level N, as illustrated in Figure 5. It analyzes `GCW<12:0>`, searching for specific bit sequences.

The circuit-level design of the GCW is depicted in Figure 11(b). It comprises two multiplexers 3-to-1 (M1 and M2), which share the same selection signal (`sel<0:1>`). The signal `sel<0>` is directly connected to `GCW<12>`, while `sel<1>` is the output of a 4-input NOR gate (`GCW<11:8>`). When `sel<0>` = 0, the weight value is zero, encoded using a 1-bit code. Instead, when`sel<0>` = 1, `sel<1>` controls the output of the multiplexers. If the `GCW<11:8>` bits are different from "0000", they represent a small-magnitude weight encoded using a 5-bit code. The corresponding value is GCW<11:8>, sign-extended to N bits (`SigExt`). Finally, if `GCW<11:8>` = "0000", the code-word represents a large-magnitude weight, whose value is encoded in the `GCW<7:8-N>` bits using a variable code length that varies from 9 to 13.

Figure 11(c) illustrates an example of run-time decoding considering $N = 6$. In the first one, `GCW<12>` = 0. Consequently, the weight value is zero, encoded using 1 bit. Thus, the shift register shifts one position, allowing the next weight to be decoded. Next, in the second example, `GCW<12:8>` = "10110". The GCW decoder extracts the last 4 bits and concatenates them with "00" to form a 6-bits binary value. The third example shows a similar case, but, as the read value is negative, "11" is concatenated. The last example depicts the case where the GCW decoder finds the sequence "10000". The binary value is then retrieved in `GCW<7:2>`.

6.3 BC instruction decoder

The BC instruction decoder converts the weights' bits into BC instructions, defining the `RightShift`, the `Add`, and the `2sComp` signals. It also governs the write back signal to store the result of in-memory operations in the BC arrays.

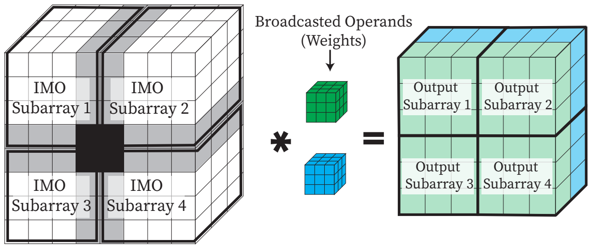


Figure 12. Convolutional layer activation tilling and transferred to different subarrays, while weights are converted into BC operations.

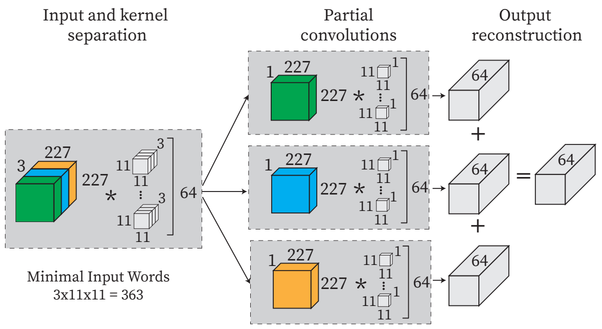

Input and kernel separation
Partial convolutions
Output reconstruction
Minimal Input Words 3x11x11 = 363

Figure 13. Example of convolution layer inputs that surpass the capacity of a subarray, this layer is deployed with partial convolutions, which requires an extra in-memory operation to reconstruct the output activation.

The sequence of shift-add operations implementing a MAC is entirely skipped when a "0" weight value is decoded, which results in energy and performance gains, as discussed in Section 9.

7 Mapping CNNs to BC Arrays

The presented BC architecture effectively accelerates the execution of convolutional and fully connected layers, whose workload dominates the execution of CNN models (e.g., they constitute more than 98% of the run-time in the benchmarks in Section 8). When deploying a CNN layer onto a BC array of given dimensions, primary goals are the reduction of data transfers between the subarrays and the periphery, and the maximization of parallelism when computing MACs. To this end, we developed an automated operations scheduler, which distributes workloads to the BC architecture considering hardware constraints, such as the number of available subarrays and their size, as well as application characteristics, including the type of CNN layer (convolutional or fully connected) and its geometry. Because large CNN layers may not entirely fit the limited memory size of the available BC arrays, the operations scheduler divides each layer into smaller blocks, or tiles, processed in parallel by each subarray. We detail next the specific mapping strategy for either CONV and FC layers.

7.1 Convolutional Layers

As presented in Table 1, for convolutional layers activations are in-memory operands, while weights are broadcasted to subarrays in the form of BC instructions. Figure 12 shows an example of how the layers are split into tiles and assigned to subarrays. In this example, input activations consist in a matrix of 8x8x3 that is convoluted by two filters of 3x3x3. Each output element is generated by computing the Hadamard product, or element-wise product, between a filter and a fraction of the input matrix having the same size, and summing up all the elements of the output Hadamard product. Sliding the filter over the entire input matrix allows the evaluation of different output elements. Hence, the convolution of the input features with one filter results in a bi-dimensional output (of size 6x6x1 in the example). Then, multiple filters compute different convolutions, producing a three-dimensional output (6x6x2 Figure 12).

To parallelize this computation pattern, a filter is used to convolve multiple input memory regions at the same time, storing the data pertaining to each region as the in-memory operands of different subarrays. Note that this parallelization strategy involves broadcasting the filter weights to subarrays. A subdivision in four memory regions is depicted in the example in Figure 12, each computing a quarter of the output (a 3x3x2 matrix). Due to border effects, memory regions must partially overlap. In the example, the elements in grey are transferred to two subarrays, while the elements in black are transferred to all of them. For practical subarray sizes, nonetheless, this effect is marginal, even if the size of tiles reduces when the number of employed subarrays increases.

For large CONV layer sizes and small subarray size, even small tiles require more input words to compute a single output value than the subarray storage capacity. For example, considering the first layer of AlexNet that has 64 filters of size 11x11x3, requiring 363 input words to calculate a 1x1 output for all the 64 filters (1x1x64), which surpasses the BC implementation detailed in Section 8 (320 16-bits words per subarray). In these cases, partial convolutions are performed, as shown in Figure 13. Consequently, filters are decomposed in the depth direction. Then, partial convolution results are retrieved with BC operations as before. Finally, the outputs of the partial convolutions are merged with additional in-memory operations.

7.2 Fully Connected Layers

Fully connected layers compute Y outputs from X inputs by performing a vector-matrix multiplication with a $X \times Y$ weight matrix W. In contrast to CONV layers, each weight is only used once at each inference in FC ones. Therefore, it cannot be used as a broadcasted operand. Instead, inputs do exhibit data reuse, as all X components are employed in the computation of each output. Therefore, we store weights as IMOs in memory subarrays for FC layers, and employ inputs as BOs.

This mapping is exemplified in Figure 14(a), where the input vector X has four elements and output Y has three elements. Consequently, twelve weights are required for the layer in the example. Figure 14(b-c) depicts how this layer is mapped into three subarrays to calculate the three outputs (i.e., one output per subarray). The MAC operations are performed between the common input X, which is broadcasted to the subarrays as BC instructions, and the stored weights W. Thus, all the three outputs are calculated in parallel.

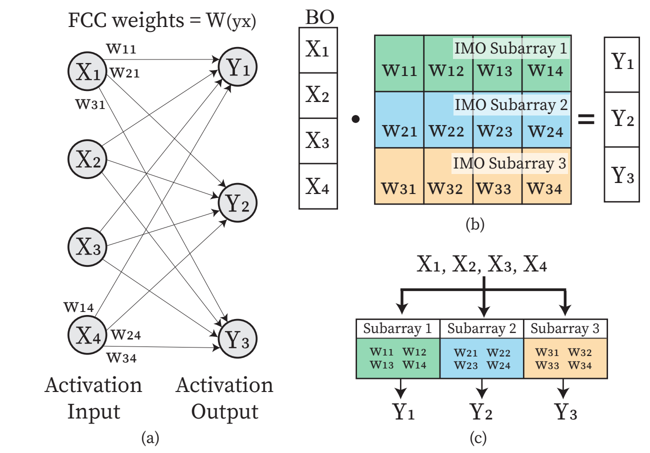


Figure 14. Example of FC layer represented in (a) graphical form and (b) matrix multiplication. (c) Data mapping to execute this workload on a BC array.

Such FC mapping is not amenable to data encoding strategies such as GCW, because broadcasted operands are activation values computed at run-time during inference. Nonetheless, FC layers exhibit usually smaller workloads than CONV ones and are not, in general (as in the benchmarks considered in Section 9), run-time bottlenecks.

8 Experimental Setup

8.1 Evaluation on CNN benchmarks

We evaluate our architecture and optimization framework on several edgeAI benchmarks of different complexity to demonstrate the effectiveness of our approach in a significant range of applications: we consider LeNet-5 [27] on CIFAR-10 [28] and AlexNet [29], VGG16 [30], MobileNet [31] and Xception [32] on the CIFAR-100 dataset [28]. Accuracy values for various CNNs and optimization levels are retrieved using PyTorch [33] and the quantization functions described in [34].

CNNs are first trained using floating-point precision for 200 epochs, obtaining accuracies in line with the state-of-the-art. Similar to [8], models are then homogeneously quantized to 16-bits in-memory operands and 8-bits broadcasted operands, and refined for 20 additional training epochs. This configuration, which has no accuracy loss with respect to employing floating-point weights and activations, is assumed as the starting point for further optimizations using the proposed methodology.

To establish a baseline, we iteratively repeat quantization and retraining to homogeneously reduce, down to 2 bits, the bitwidth of the operands streamed into the subarrays. Then, we retrain the models for five fine-tuning epochs at each step. Five epochs are also run when applying our heterogeneous approach, as described in Section 4, after each BOs and IMOs optimization steps (phases (b) and (d) in Figure 3).

8.2 Circuit-level characterization

As a test vehicle for our experiments, we consider a BC architecture composed of a varying number of subarrays. Each subarray contains 5 LGs of 32 rows each, totalling 160 memory rows. Each row is composed of 2 interleaved 16-bits words. Therefore, the subarray stores 5120 bits (640 bytes). From a layout perspective (Figure 15(a)), the memory cell array is organized into 160 rows and 32 columns. It can store 320 words in 1x16bit mode or 640 words in 2x8bit mode. Using a methodology analogous to [5], we implement the BC architecture as a full-custom design and performed HSpice energy and timing characterization. Targeting a 28nm CMOS technology from TSMC, the architecture can operate at a maximum frequency of 2.2GHz. The subarray requires 376pJ for reading a 16-bit word, and 414pJ energy for writing it. An in-memory shift-add operation requires 381pJ. The subarray has an area of $1240\mu m^2$, of which 26.5% is used for the BCU and the LGP circuits. 6.5% of the total area is used to implement the negation, embedded shift, and wordline parallelism. Finally, 67% of the area is filled up by the SRAM cells.

The GCW decoder is designed as semi-custom IC, as shown in Figure 15(b). It is synthesized, placed and routed (again, in 28nm CMOS technology from TSMC) to extract its area, timing, and energy requirements. The circuit has an area of $760\mu m^2$, which represents 61% of the area of a single subarray, and less than 1% of the total area in a 128 subarrays configurations. The decoder requires 1fJ per cycle to operate.

We compare performances with the subarray design described in [5]. Such architecture has been shown to achieve 3x better performance, and 1.5x increased energy efficiency compared with the ARM NEON SIMD accelerator when running inferences on benchmark CNNs.

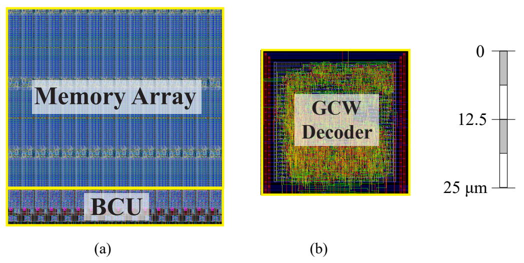


Figure 15. Layout of (a) semi-custom design of the decoder circuit and, (b) full-custom design of one memory subarray.

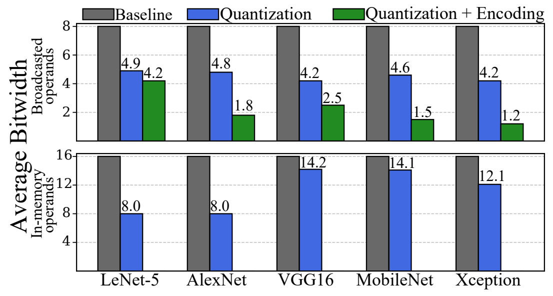


Figure 16. Average bitwidths achieved in our evaluated benchmarks by employing a synergic use of heterogeneous quantization (blue bars) and GCW encoding (green bars), for maximum accuracy degradation constraint of 1%.

9 Results

9.1 Accuracy-constrained compression

Figure 16 depicts the average bitwidth (across layers) of IMOs and BOs in CNN benchmarks applications optimized with our proposed methodology. In all cases, results are for an accuracy threshold of 1% with respect to implementations having 16-bits IMOs and 8-bits BOs.

Blue bars illustrate the average bitwidth reduction achieved in convolutional and fully connected layers by mean of our heterogeneous quantization approach (as detailed in Section 4.1), which results in compression ratios of CNN models of 76.8% on average.

Then, additional savings are achieved by encoding the weights of convolutional layers (illustrated in Section 4.2). Results are shown as green bars in Figure 16. They show that GCW encoding effectively contributes to the reduction of storage requirements, resulting in overall model size savings of 85.3% on average.

Experimental outcomes show that all activations of convolutional layers of simpler CNNs (LeNet-5 and AlexNet) can be effectively reduced to 8-bits while abiding by the accuracy constraint. Such optimization instead can only be selectively applied in more complex benchmarks such as in VGG16, MobileNet and Xception, highlighting the benefit of a heterogeneous approach.

Moreover, especially high compression ratios are achieved for larger models, because their size is largely determined by the footprint of convolutional weights. These can be effectively compressed by quantization and encoding (by up to more than 20x for the Xception CNN).

9.2 CNN Inference Cycle-count Reduction

Figure 17 illustrates the accuracy/performance trade-off in different optimized benchmarks. Black markers report the accuracy of homogeneously quantized models, while blue and green lines show the accuracy achieved in the various steps of the proposed hardware/software co-design methodology, for accuracy degradation thresholds of 1% and 5%, respectively, compared to configurations with 8-bit BOs and 16-bits IMOs.

In Figure 17, the points (b), (c) and (d) highlight improvements obtained in the different stages of the optimization strategy, as illustrated in Figure 3 and detailed in Section 4. They report performance/accuracy of CNNs after (b) BOs, (c) convolutional filters, and (d) IMOs optimization.

Points (e) and (f) report further cycle-count reductions obtained by hardware optimizations. In (e) up to three single-cycle bit-shifts (NES = 3 in Section 5.3) are supported. Additionally, in (f), MAC operations involving zero-valued broadcasted operands are skipped (Section 6).

When employing 5 bits, baseline uniform quantization achieves a 33% cycle-count reduction at the cost of an average 1.3% accuracy degradation, which rapidly increases for smaller bitwidths. Conversely, significantly higher performance improvements are obtained with our approach. In particular, heterogeneous quantization alone (steps (b)-(d)) enables alone up to 80% cycle-count gains. BC hardware optimizations are also highly effective (steps (e) and (f)). Support for NES=3 results in an average cycle-count reduction of 2.1x, which increases to 2.9x when also skipping multiplications involving zero-valued broadcasted operands. Considering all software and hardware optimizations, the co-design framework achieves an average cycle-count reduction of 89.3% (a speed-up of 11.5x) for 8-bits quantized

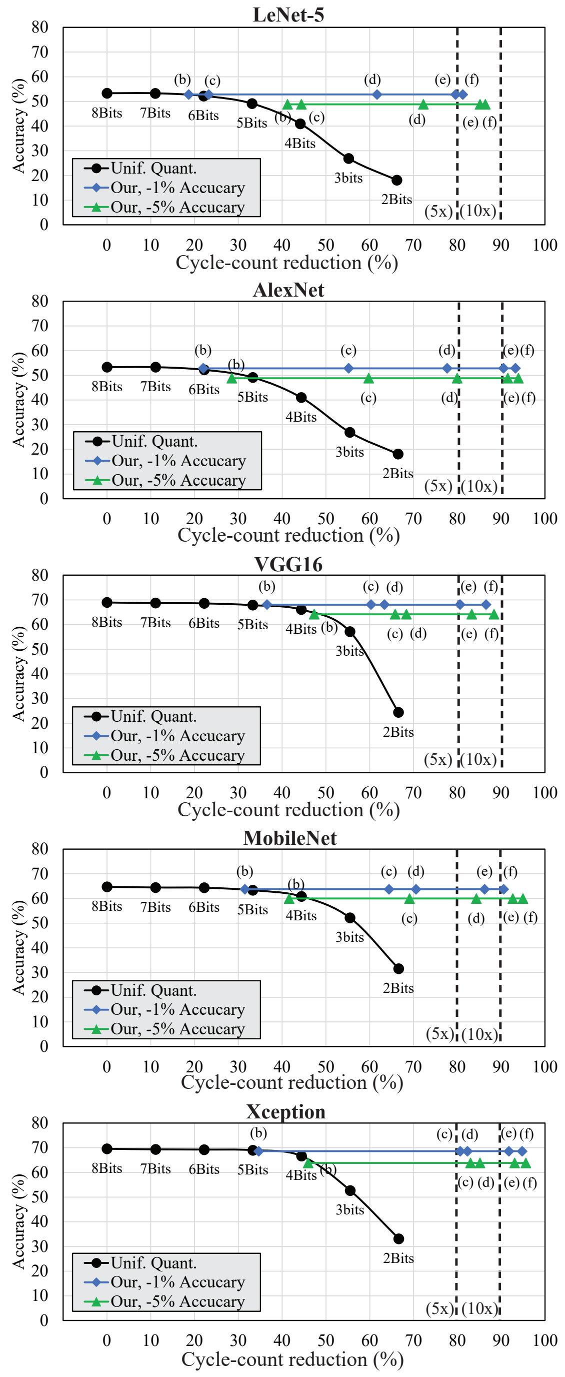


Figure 17. Accuracy and cycle-count reductions in homogeneously quantized CNNs (black lines) and optimized CNNs (blue and green lines) for a 1% and a 5% accuracy drops. Data refers to single-subarray BC architectures. Vertical dashed lines mark speed-up levels of 5x and 10x.

baselines for 1% degradation thresholds, and 91.9% average cycle-count reduction (a speed-up of 15x) for 5% accuracy degradation.

Performance gains are linearly correlated to energy savings, as they derive from reductions in shift-adds. Illustrating this aspect, Table 2 shows the Inferences Per Second (IPS) and inference energy in 8-bits quantized CNN baselines and in optimized models. As an example, the baseline AlexNet requires a 9.16mJ to perform one inference. Our optimized hardware and software reduce this cost by 93%, to just 0.62mJ, when executing on a single subarray.

Table 2. Inferences-per-second (left) and inference energy (right) in homogeneously quantized implementations executing on the BC architecture in [5] compared to heterogeneously quantized models executed on optimized BC architecture and employing 1, 32 or 128 subarrays. Optimized implementations are obtained for a 1% accuracy degradation threshold.

	Inferences Per Second (IPS)				Energy (mJ)			
	Simon *et al.* [5]	**Optimized (1% drop)**			**Simon *et al.* [5]**	**Optimized (1% drop)**		
		1 sub	**32 subs**	**128 subs**		**1 sub**	**32 subs**	**128 subs**
LeNet-5	289	1415	22491	22545	0.0035	0.0007	0.0012	0.0045
AlexNet	0.11	1.62	36	73	9.16	0.62	0.64	0.71
VGG16	0.39	2.89	78	201	2.57	0.35	0.35	0.39
MobileNet	0.35	3.76	93	220	2.82	0.27	0.27	0.29
Xception	0.0023	0.09	2.18	5.13	432.69	11.34	11.54	12.26

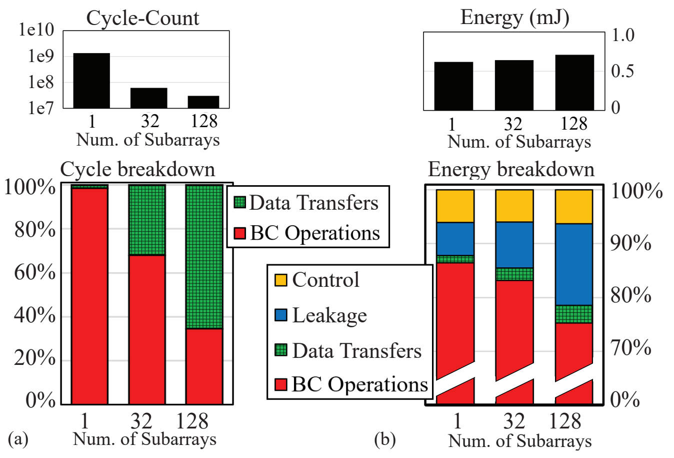


Figure 18. (a) Cycle and (b) energy breakdown of multiple subarray architecture for an AlexNet inference with SW-HW optimization.

9.3 BC Architectures with Multiple Subarrays

As shown in Table 2, the IPS of our evaluated benchmarks scales up with the number of subarrays, as the workload is effectively distributed using the CNN mapping strategy proposed in Section 7. For AlexNet, VGG16, MobileNet, and Xception, on average a speed-up of 58x is reached when employing 128 subarrays with respect to a single one. Being a smaller network, LeNet-5 is less amenable to parallelization, reaching in this setting a 15x speed-up.

The performance of our solution when varying the number of subarrays are further detailed in Figure 18(a) for the Alexnet benchmark. In its top part, the figure reports absolute cycle-counts and energy requirements. Below, we show the proportional breakdowns. These results indicate that, when the workload is entirely run on a single subarray, 99% of the clock cycles are used to perform MAC operations. However, the use of 32 or 128 subarrays reduces this percentage to 68% and 35% (respectively) because data-transfers are performed sequentially, while the parallelism of BC operations grows linearly with the number of subarrays.

Then, the energy results included in Table 2 show that the use of a high number of subarrays has only a small impact on energy efficiency (while greatly reducing runtimes). For AlexNet, VGG16, MobileNet, and Xception the difference in energy between 1 and 128 subarray configurations is only 12% (again, LeNet-5 is an outlier due to its small dimensions). Indeed, the number of subarrays does neither influence the energy cost of BC operations nor the number of BC operations required by an inference. The slight increase in energy consumption is due to leakage energy (Figure 18(b)) consumed by subarrays during data transfers, and by the larger and more energy-hungry H-tree required to connect them. On the other hand, a higher level of parallelism requires decoding a smaller number of BC instructions, decreasing the related energy budget, which nonetheless accounts for less than 1% of the total energy in all configurations.

10 Conclusion

Edge AI requires the execution of extremely computational and memory-intensive applications on constrained platforms. Bit-Line computing is a promising avenue to cope with this challenge, but demands careful synergic co-optimization of applications and hardware. To tackle this problem, in this work we have presented a framework that comprises hardware-aware application optimizations as well as novel architectural solutions to effectively harness them. On the application side, CNNs are compressed by combining heterogeneous quantization and weights encoding. In turn, the proposed Bit-line Computing (BC) platform embeds low-overhead hardware features to perform runtime decoding, support fine-grained quantization of broadcasted operands, and leverage word-level parallelism of in-memory operands. Results on a variety of CNN benchmarks have demonstrated that, for a 1% accuracy degradation constraint, our compression strategy offers 85% average memory reductions compared to uniformly quantized CNN implementations. Using the same constraint, our proposed BC architecture achieves on average 11x inference speedups and 90% energy savings with respect to state-of-the-art BC approaches.

Acknowledgments

This work has been supported by the EC H2020 WiPLASH (Ga. No. 863337), the EC H2020 FVLLMONTI (Ga. No. 101016776), the Swiss NSF ML-Edge (Ga. No. 182009), and the ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

References

[1] A. Khan *et al.*, “A survey of the recent architectures of deep convolutional neural networks,” *Springer Artificial Intelligence Review*, 2020.

[2] X. Wang *et al.*, “Convergence of edge computing and deep learning: A comprehensive survey,” *IEEE Communications Surveys & Tutorials*, 2020.

[3] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE JSSC*, 2016.
[4] D. Rossi *et al.*, "PULP: A parallel ultra low power platform for next generation IoT applications," *IEEE HCS*, 2015.
[5] A.-W. Simon *et al.*, "BLADE: An in-cache computing architecture for edge devices," *IEEE Transactions on Computers*, 2020.
[6] M. Rios *et al.*, "Running efficiently CNNs on the edge thanks to hybrid SRAM-RRAM in-memory computing," *IEEE/ACM DATE*, 2021.
[7] ——, "An associativity-agnostic in-cache computing architecture optimized for multiplication," *IEEE VLSI-SoC*, 2019.
[8] F. Ponzina *et al.*, "A flexible in-memory computing architecture for heterogeneously quantized CNNs," *IEEE ISVLSI*, 2021.
[9] A. Garofalo *et al.*, "Darkside (2021)," http://asic.ee.ethz.ch/2021/Darkside.html.
[10] S. Bianco *et al.*, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, 2018.
[11] L. Song *et al.*, "Pipelayer: A pipelined ReRAM-based accelerator for deep learning," *IEEE HPCA*, 2017.
[12] X. Peng *et al.*, "Optimizing weight mapping and data flow for convolutional neural networks on processing-in-memory architectures," *IEEE TCAS I*, 2019.
[13] J. Rhe *et al.*, "VW-SDK: Efficient convolutional weight mapping using variable windows for processing-in-memory architectures," *arXiv preprint arXiv:2112.11282*, 2021.
[14] S. Han *et al.*, "Learning both weights and connections for efficient neural network," *NIPS*, 2015.
[15] Y. He *et al.*, "Learning filter pruning criteria for deep convolutional neural networks acceleration," *Proceedings of the IEEE/CVF*, 2020.
[16] P. Molchanov *et al.*, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.
[17] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," *IEEE/ACM DAC*, 2018.
[18] B. Denkinger *et al.*, "Impact of memory voltage scaling on accuracy and resilience of deep learning based edge devices," *IEEE Design & Test*, 2019.
[19] A. Jain *et al.*, "Symmetric k-means for deep neural network compression and hardware acceleration on FPGAs," *IEEE JSTSP*, 2020.
[20] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.
[21] J.-H. Lee *et al.*, "Arithmetic coding-based 5-bit weight encoding and hardware decoder for CNN inference in edge devices," *IEEE Access*, 2021.
[22] D. Ielmini *et al.*, "Device and circuit architectures for in-memory computing," *Wiley Advanced Intelligent Systems*, 2020.
[23] ——, "In-memory computing with resistive switching devices," *Nature Electronics*, 2018.
[24] A. Levisse *et al.*, "Architecture, design and technology guidelines for crosspoint memories," *IEEE/ACM NANOARCH*, 2017.
[25] S. Jeloka *et al.*, "A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6t bit cell enabling logic-in-memory," *IEEE JSSC*, 2016.
[26] K.-C. Akyel *et al.*, "DRC2: Dynamically reconfigurable computing circuit based on memory architecture," 2016.
[27] Y. Lecun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.
[28] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," *University of Toronto*, 2009.
[29] ——, "Imagenet classification with deep convolutional neural networks," *NIPS*, 2012.
[30] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.
[31] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv:1704.04861*, 2017.
[32] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *Proceedings of the IEEE/CVF*, 2017.
[33] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv:1912.01703*, 2019.
[34] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *Proceedings of the IEEE/CVF*, 2018.

Marco Rios received the M.Sc. degree in Computer Science and Electronics For Embedded Systems from Université Grenoble Alpes, France, in 2018. He is currently a PhD student at the Embedded Systems Laboratory of EPFL, Switzerland. His research interests include design of integrated systems and circuits, in-SRAM computing and the system impact of emerging memories.

Flavio Ponzina received the M.Sc. degree in Computer Engineering from Politecnico di Torino, Italy, in 2018. He is currently a PhD student at the Embedded Systems Laboratory (ESL), EPFL. His main research interests include low power architectures and AI-based systems optimization.

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

Giovanni Ansaloni is a senior researcher at the Embedded Systems Laboratory of EPFL (ESL-EPFL, Lausanne, CH). He previously worked as a Post-Doc at the University of Lugano (USI, CH) between 2015 and 2020, and at EPFL between 2011 and 2015. He received a Ph.D. degree in Informatics from USI in 2011. His research efforts focus on domain-specific and ultra-low-power architectures and algorithms for edge computing systems, including hardware and software optimization techniques.

David Atienza David Atienza (M'05-SM'13-F'16) is professor of electrical and computer engineering, and head of the Embedded Systems Laboratory (ESL) at EPFL, Switzerland. He received his PhD in computer science and engineering from UCM, Spain, and IMEC, Belgium, in 2005. His research interests include system-level design methodologies for high-performance multi-processor system-on-chip (MPSoC) and low power Internet-of-Things (IoT) systems, including thermal-aware design for MPSoCs and many-core servers, and edge AI architectures for wearables and IoT systems. He is a co-author of more than 400 papers, one book and 14 patents in these fields. He served as DATE General Chair and Program Chair, and is currently Editor-in-Chief of IEEE TCAD (period 2022-2023). Among others, Dr. Atienza has received the ICCAD 2020 10-Year Retrospective Most Influential Paper Award, the 2018 DAC Under-40 Innovators Award, an ERC Consolidator Grant in 2016, the 2013 IEEE CEDA Early Career Award, and the 2012 ACM SIGDA Outstanding New Faculty Award. He is a Fellow of IEEE, a Fellow of ACM, and served as IEEE CEDA President (period 2018-2019).