Overflow-free compute memories for edge AI acceleration

FLAVIO PONZINA, MARCO RIOS, ALEXANDRE LEVISSE, GIOVANNI ANSALONI, and DAVID ATIENZA, École Polytechnique Fédérale de Lausanne (EPFL), Embedded Systems Laboratory, Switzerland

Compute memories are memory arrays augmented with dedicated logic to support arithmetic. They support the efficient execution of data-centric computing patterns, such as those characterizing Artificial Intelligence (AI) algorithms. These architectures can provide computing capabilities as part of the memory array structures (In-Memory Computing, IMC) or at their immediate periphery (Near-Memory Computing, NMC). By bringing the processing elements inside (or very close to) storage, compute memories minimize the cost of data access. Moreover, highly parallel (and, hence, high-performance) computations are enabled by exploiting the regular structure of memory arrays. However, the regular layout of memory elements also constrains the data range of inputs and outputs, since the bitwidths of operands and results stored at each address cannot be freely varied. Addressing this challenge, we herein propose a HW/SW co-design methodology combining careful per-layer quantization and inter-layer scaling with lightweight hardware support for overflow-free computation of dot-vector operations. We demonstrate their use to implement the convolutional and fully connected layers of AI models. We embody our strategy in two implementations, based on IMC and NMC, respectively. Experimental results highlight that an area overhead of only 10.5% (for IMC) and 12.9% (for NMC) is required when interfacing with a 2KB subarray. Furthermore, inferences on benchmark CNNs show negligible accuracy degradation due to quantization for equivalent floating-point implementations.

CCS Concepts: • **Computing methodologies** → **Machine learning algorithms**; • **Hardware** → **Emerging architectures**; **Hardware-software codesign**; • **Computer systems organization** → **Embedded systems**.

Additional Key Words and Phrases: In-Memory Computing, Near-Memory Computing, Edge Machine Learning, Quantization, Convolutional Neural Networks.

1 INTRODUCTION

Edge Artificial Intelligence (AI) is fostering a revolution in a multitude of scenarios, ranging from connected health-care [12] to autonomous navigation [19]. The main challenge facing edge AI comes from the tension between the high computational and memory requirements of AI models and the paucity of resources available at the edge. To address it, the co-optimization of applications and hardware is key, because large efficiency benefits can be reaped by exposing opportunities at the algorithmic level and harnessing them with the design of specialized hardware.

Edge AI applications are mostly data-centric, requiring a large amount of energy-hungry data transfers, which can consume orders of magnitude more energy than arithmetic operations [8]. To ease this burden, compute memories enrich memory arrays with distributed Processing Elements (PEs), capable of executing arithmetic operations in place, challenging the separation of storage and computation at the bases of classic Von Neumann architectures [30][3].

Compute memories can be classified into In-Memory Computing (IMC) and Near-Memory Computing (NMC). In IMC approaches [37], the memory array itself is employed to perform computations, usually exploiting the concurrent read access of multiple memory cells connected to the same sense amplifier. In contrast, NMC solutions [33] only instantiate digital logic gates to support arithmetic operations. IMC can achieve high area and energy efficiency, as it takes advantage of and reuses standard memory components, such as read-out circuits. On the other hand, it mandates a higher design effort with respect to NMC, as the latter does not require any modification of the design of memory arrays, but only deals with the peripheral circuitry. Focusing on SRAM technology, we propose and contrast in this paper implementations of both paradigms, evaluating their respective benefits. We discuss compute memory solutions based on different bit-cell technologies in Section 3.

In both NMC and IMC, the regular structure of memories is leveraged to allow the parallel execution of multiple operations, as memories are typically divided into subarrays, each having its own periphery and controller. This arrangement opens exciting opportunities for the in/near-memory execution of AI applications, as these are characterized by a high degree of parallelism and operand reuse [26]. In particular, parallel dot-vector operations (e.g., implementing convolutions) can be executed in memory with very little additional hardware. Indeed, dot-vectors can be decomposed in a series of shift-add operations between in-memory operands and accumulators (local to each subarray), governed by the bits of broadcasted operands, common across all subarrays.

The above-mentioned computation paradigm paves the way for the fine-grained quantization of broadcasted operands, as modulating their bitwidth governs a trade-off between accuracy and performance. Previous works have investigated the opportunities offered by low bitwidth representations for the optimization of convolutional neural networks (CNNs) runtimes [38] [10] [13]. Nevertheless, these works consider quantization when targeting dedicated hardware accelerators. Complementing our contribution to hardware design aspects, this work instead presents a per-layer, accuracy-driven methodology dedicated to compute memories.

While quantization restricts the bitwidth of inputs, subsequent computations (e.g., multiplications, accumulations) can typically increase the value ranges of intermediate results and outputs. This issue is particularly detrimental for compute memories because memory arrays require that all stored values are within well-defined bounds. A naive implementation may either incur overflows (which catastrophically affect accuracy) or require many bits to sign-extend operands (wasting memory resources).

We show that the above-mentioned challenge can be addressed by hardware/software co-design, i.e., by carefully tailoring the data representation during software optimization, and providing domain-specific in-/near-memory operations for AI applications. In our designs, *multiplications* are executed with no overflows by construction, so that approximations are only due to the truncation of the least significant bits (with negligible impact on accuracy). The increase in data range during *accumulations* is, instead, managed in hardware, by dedicated overflow registers. The related resource overhead can be effectively managed, as the overflow register size is only logarithmically related to the number of accumulation operations. Combining these strategies, parallel and safe-by-construction dot products can be executed in compute memories, thus supporting the acceleration of a vast range of kernels in the machine learning and deep learning domains. In this work, we focus on the acceleration of convolutional and fully connected layers of CNN models, as they are widely used and intensive machine learning computational patterns. Nevertheless, any algorithm employing matrix-vector operations (as is the case in machine learning applications) can benefit from the proposed hardware design to increase computing efficiency.

Our contributions are summarized as follows:

- We showcase a new hardware/software co-design solution, based on the combination of operands scaling and the use of overflow registers. Our approach leads to safe-by-construction and efficient NMC/IMC designs devoted to hosting the computation of convolutional and fully connected layers of CNNs.
- We present two circuit-level embodiments of our approach, based on IMC and NMC, respectively. We detail their implementation and compare performance and resource requirements.
- We introduce an accuracy-driven per-layer quantization framework for AI applications, as well as inter-layer operands scaling methodology, devoted to compute memories.

- On a collection of standard CNN networks and datasets, we demonstrate that our strategy outperforms state-of-the-art alternatives in terms of achieved accuracy, with limited resource overheads of just 10.5% and 12.9% for IMC and NMC, respectively, with respect to baseline SRAM memory arrays.

The rest of the paper is organized as follows: first, we present the related background in Section 2 and include a literature review on NMC/IMC designs in Section 3. Then, in Section 4, we present our algorithmic solution to handle value ranges. Next, in Section 5, we describe how CNN models are optimized by heterogeneously tuning the bitwidth of weights and activations in different layers. In Section 6, we present the digital computing subarray and the analogue computing base subarray that implement the proposed strategy, supporting the efficient execution of the operations of dot products. Our experimental framework is then described in Section 7, and the results are presented in Section 8. Finally, we draw our conclusions in Section 9.

2 BACKGROUND

2.1 Quantization in edge AI

Quantization methodologies restrict the representable values of a datum from an initial large set (e.g., 32-bit floating-point representation) to a smaller one. In the general case, it hence involves an approximation of the datum being represented, which may hamper application-wide Quality of Service (QoS) metrics, such as accuracy. On the other side of the coin, quantization can be harnessed as an optimization opportunity, as operations on aggressively quantized values can be supported with a paucity of hardware resources. Quantization is particularly of interest in the AI domain, where computations are robust towards perturbations, and often approximate results (e.g., for object classification tasks) are acceptable [23][20].

Indeed, highly aggressive homogeneous [13] or heterogeneous [21] quantization schemes have been proposed in the literature. In the context of convolutional networks, homogeneous quantization aims at reducing the bitwidth of weights and activations of these models from the baseline floating-point format in which parameters are usually trained, to more compact integer or fixed-point representations. In particular, common quantization levels employ 8-bit weights and either 8-bit or 16-bit activations, showing a limited or negligible impact on model accuracy [23]. An extreme example of such an approach is represented by Binarized Neural Networks (BNNs), where the data is restricted to assume binary values (i.e., either 0, or 1), thus employing only one bit for their representation [10]. On the one hand, arithmetic operations are extremely simplified, as additions and multiplications can be converted into much simpler OR and XOR instructions, respectively. On the other hand, a significantly higher number of parameters is usually required to achieve accuracies comparable with floating-point CNN alternatives.

In contrast, heterogeneous quantization is an optimization strategy that takes advantage of the different degree of robustness among CNN layers to quantize more aggressively those that are more resilient against perturbations [4]. As such, different quantization levels are allowed in different CNN layers, allowing a more fine-grained optimization of CNN models, where quantization can be pushed to its limit while preserving a target QoS [21].

However, most of the works on the quantization of edge AI applications assume that only input activations and weights are quantized to a given bitwidth [11]. Such an assumption, known as “fake quantization”, is reasonable when targeting general-purpose processors, but does not hold in the case of compute memories, in which *every* stored value, including inputs, intermediate results, and outputs must abide by a given bitwidth. The fake quantization approach involves adding a quantization layer right before convolutional and fully connected layers to simulate the effects of quantization, without actually changing the format of weights and activations as stored in memory, but only constraining them to assume only the values that can be represented in a given quantization level. In this way, ordinary backpropagation methods can be used for training, while simulating the effect of quantization during the forward pass.

Notably, this approach does not perform fixed-point or integer arithmetic. Crucially, in the context of our contribution, it hence does not account for the strict limitations in the dynamic range of fixed-point formats along a chain of computations, but only at its beginning and at its end. Indeed, operations (e.g., the accumulation of accumulated products) in fake quantization are not affected by data range limitations and the ensuing overflows or value saturations.

Data range considerations must instead be carefully taken into account when targeting true fixed-point arithmetic, as we do in this work. We discuss them, for the cases of convolutional and fully connected layers, in the following.

2.2 Dot products computation

We focus on matrix-vector multiplication operations as the computational pattern of interest because they can efficiently implement both fully connected and convolutional layers. The im2col algorithm [36] is a widely used approach to transform a set of convolutional filters into a matrix, so that the computation of a convolutional layer is reduced to a matrix-vector operation. Furthermore, weights in a fully connected layer are naturally represented as matrices, where $w_{i,j}$ is the weight connecting the i-th input to the j-th output.

Matrix-vector multiplications can themselves be decomposed into dot products, where each dot product returns an element of the output vector from a row of the matrix and the vector, which must be of equal length. Since each dot product is computed independently, we focus on this computing pattern for the rest of this section. Mathematically, dot products are expressed as:

$$y = \sum_{i=0}^{V} x_i \cdot w_i \tag{1}$$

where x and w are two vectors of length V.

In a quantized form, these are expressed with a fixed number of integer and fractional bits, indicated as $QI.F$ fixed-point notation, where I is the number of bits representing the integer part (including the sign bit for signed numbers), and F is the number of bits representing the fractional one.

The bitwidth required to exactly represent the output of a single product operation is the sum of the bitwidth of multiplicands and multipliers:

$$width(x_i \cdot w_i) = width(x_i) + width(w_i) \tag{2}$$

Nevertheless, it is possible to impose smaller bitwidths for scalar multiplication outputs, by employing data truncation (i.e., by dropping the least significant bits of the result), as long as truncation only affects fractional bits. As we show in Section 8, this approximation has a negligible impact on the quality of the output for AI inference.

Crucially, truncation can constrain the output of scalar multiplication (and, as shown in Section 4, that of all intermediate values required for its computation) to have the same width as that of one of the operands, so that both can be stored in memory words of the same size in a compute memory array.

When performing dot products, the output of scalar multiplications is accumulated, as shown in Equation 1. Accumulation also has an effect on the data range of the output. In fact, the width required to perform the operation without any approximation and to avoid overflows is:

$$width(y) = width(x_i) + width(w_i) + log_2(V) \tag{3}$$

where V is the number of added elements (i.e., the length of vectors x and w). In a naive implementation, the avoidance of overflows would severely restrict the admissible data range of input operands. As an example, a 16-bit dot product output of two 8-element input vectors x and w (a very small number in practice) can be safely computed without any truncation or hardware support only by restricting the input operands range to 6 bits, because the multiplication of two 6-bit numbers requires 12 bits and $log2(8) = 3$ additional bits are required to avoid overflow during accumulation. In such a naive strategy, inputs would then have to be quantized to small bitwidths (e.g., 6 bits) and then sign-extended to the memory word size (e.g., 16 bits), greatly hampering QoS while wasting memory resources to store sign-extension bits which do not convey meaningful information.

Instead, our strategy controls the increase in data range due to multiplications and accumulations by a combination of (a) careful use of data truncation during computations, (b) pre- and post-scaling operands to abide by a proper fixed-point notation when in-memory, so that truncations only ever affect fractional bits, and (c) by employing dedicated registers to accommodate the data range increase when performing accumulations. As an example, a 16-bit memory can convolve input activations encoded as 16-bit numbers, without overflow.

3 RELATED WORKS ON COMPUTE MEMORIES

The high computational demands of AI applications strain the capabilities of traditional Von-Neumann architectures, especially when considering the tight resource constraints present in edge AI scenarios. The ensuing challenge has fostered a renewed interest in domain-specific computing solutions, inspiring novel approaches based on a variety of approaches based on FPGAs, GPUs, systolic arrays, etc. [24] [1], arguably generating a "Cambrian explosion" [34] of heterogeneous solutions in the computing architecture landscape.

In this context, compute memories are particularly promising because they leverage the data-centric nature of AI algorithms, i.e., the fact that applications are composed by manipulations on a very large number of data items. Indeed, compute memory solutions have been derived in many different ways in recent years, exploiting different technologies, such as Dynamic and Static Random-Access memories (DRAM and SRAM), and resistance-based memories (PCM and RRAM) [28]. Moreover, compute memories can be divided into two subcategories: Near-Memory Computing (NMC) and In-Memory Computing (IMC). While the former only read the memory and compute the operations independently, the latter compute (fully or partially) operations as part of data accesses. We herein reference notable works in this field, with a particular focus on SRAM-based compute architectures, which are most related to our research contribution.

3.1 Non-SRAM compute memories

Proposed DRAM compute memories usually target high-performance computing systems, as DRAM high-energy requirements and high fabrication cost hamper its usage in low-power edge AI devices. However, the DRAM fabrication process is not fully compatible with CMOS-based logic, imposing challenges on the Processing Element (PE) design. Addressing it, the authors of [17] [6] introduced a simple near-memory PE design that can be synthesized in DRAM technologies and integrated at the bank level, at the cost of high area overheads ($\sim$ 50% of the chip area is devoted to PEs in [17]) and sub-optimal performance. Instead, the authors of [5] describe a near-memory architecture in which memory and logic chips are alternated in a 3-dimensional stack. Such solutions have to cope with large and high-capacitance vertical through-silicon vias (TSVs) for logic-to-memory connectivity, limiting bandwidth and integration density.

Few in-memory DRAM approaches have also been proposed [29][42]. Nonetheless, these are limited by the destructive reads characterizing DRAM technologies, so that, to avoid data loss while performing computations, data needs to be copied elsewhere, decreasing efficiency.

Further non-SRAM architectures have instead focused on emerging Non-Volatile Memories (eNVM) technologies, such as Phase-Change Memory (PCM) and Resistive Random-Access Memory (RRAM). When integrated along

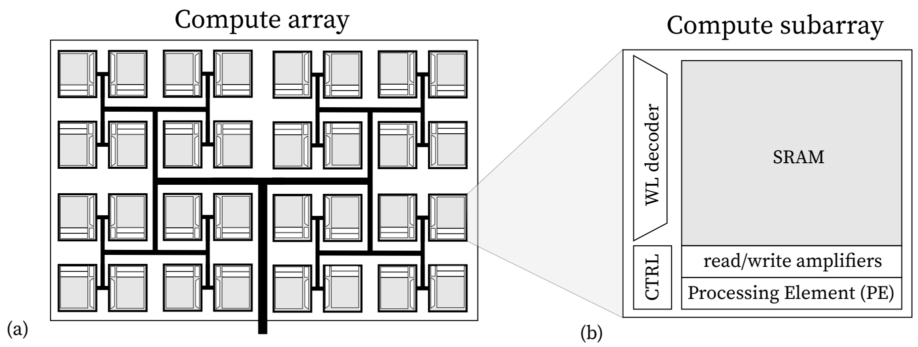


Fig. 1. (a) H-tree composition of SRAM subarrays that can compute in parallel. (b) Computing subarray composed of a single WL decoder, the subarray controller, the read/write amplifiers, and the computing unit; the last can be implemented by attaching a fully digital unit on the periphery, or by relying on the bit-line computing strategy.

analog-to-digital and digital-to-analog converters, these devices can be employed to perform in-memory matrix-vector multiplications in the analog domain by exploiting Kirchhoff's laws [2][39][14]. They are characterized by zero idle power consumption, but also present high write currents, high temporal and spatial variability, and low endurance.

3.2 SRAM-based Near-Memory Computing

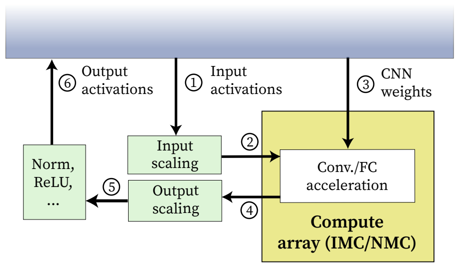
SRAM-based near-memory computing architectures leverage the regular structure of SRAM arrays. These are composed of several subarrays typically interfaced with a system bus via an H-tree interconnect (Figure 1-a). As SRAM can be readily co-integrated with CMOS logic on the same die, processing capabilities can be added as digital logic at the periphery of each subarray, hence enabling high parallelism and reducing data transfer requirements. To this end, in [22], the authors use look-up tables to accelerate execution, showing 3× speed-ups with respect to GPUs on AI algorithms.

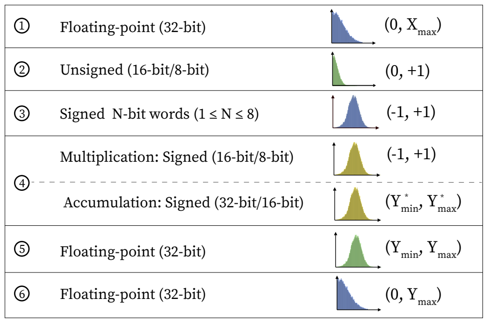
Further works in this area exploit near-memory PEs to implement domain-specific functional units. Authors of [43] introduced near-SRAM shifters, rotators, and s-boxes for cryptographic applications, while in [44] a multi-bit product-sum engine is described.

3.3 SRAM-based In-Memory Computing

In-memory strategies aim to exceed the performance of NMC architectures by adopting modified memory arrays to execute part of the implemented functionality. Most approaches in this field are based on the concurrent activation of multiple bit-cells in the same memory column, hence implementing the bit-wise AND and NOR operations between two words. At the memory array periphery, digital logic is to derive arithmetic operations from the bit-line signals. Only four additional logic gates are required to derive *sum* and *carry-out* signals for each bit position. Carries are then connected to support in-memory additions [31]. Multiplications can also be performed, employing several steps as a sequence of additions between two values, where the first one is the partial result and the second one is, at each step, either the multiplicand or the value '0', depending on a bit of the multiplier (as discussed in detail in Section 6).

While previous studies have showcased the efficiency of the bit-line computing paradigm, these architectures usually require a complex interface and decoder, as in the general case two addresses (corresponding to the two operands) must be specified. Moreover, 2 clock cycles for each in-memory operation are required, as one cycle is





①	Floating-point (32-bit)		$(0, X_{max})$
②	Unsigned (16-bit/8-bit)		$(0, +1)$
③	Signed N-bit words ($1 \le N \le 8$)		$(-1, +1)$
④	Multiplication: Signed (16-bit/8-bit)		$(-1, +1)$
	Accumulation: Signed (32-bit/16-bit)		(Y^*_{min}, Y^*_{max})
⑤	Floating-point (32-bit)		(Y_{min}, Y_{max})
⑥	Floating-point (32-bit)		$(0, Y_{max})$

Fig. 2. Acceleration of convolutional and fully connected layers. Input activations are first scaled (1) and then stored in the compute arrays as IMOs (2). Weights are instead streamed into the accelerator (3) to perform MAC operations (4). Then, outputs are scaled back (5) and finally stored outside the memory computing banks (6). The table shows the data range at each step. Focus of this work is the design of the compute memory accelerator, represented by the yellow box.

used to write back the data in memory. The proposed IMC architecture is not hampered by this issue. Moreover, previous compute memory architectures either overlooked the problem of data range bounds [18] or only focused on single arithmetic operations [25]. We instead propose an implementation able to execute entire neural network layers, safe-by-construction from overflows.

4 PRE- AND POST-SCALING FOR FULL RANGE IN-/NEAR-MEMORY COMPUTING

To avoid overflow when performing in-/near-memory computation, we rely on pre- and post-scaling of input/output activations at the boundaries of convolutional and fully connected layers. Our approach fully utilizes the available data range offered by the word size of the bit-line computing memory, minimizing the bits devoted to sign extension.

Input/output scaling has linear complexity in the data size, e.g., $O(A^2C)$ for activations with C channels of $A \times A$ size. Instead, convolutions require a number of multiply-accumulate (MAC) operations in the order of $O(A^2CK^2L)$ when L filters of size $K \times K \times C$ are used. Similar considerations can be applied to fully connected layers: their number of MAC operations is in the order of $O(IO)$, where I and O represent the number of input and output features, respectively. In contrast, the complexity for input activations scaling is just $O(I)$. The implementation of hardware executing scaling operations is outside the scope of the paper. Nonetheless, measurements on commodity hardware executing non-accelerated versions of the CNN benchmarks in Section 7 reveal that the runtime devoted to pre- and post-scaling operations is negligible in all cases, confirming the outcome of the complexity analysis outlined above. Indeed, it only accounts for 0.03% of the inference runtime, on average.

Figure 2 provides a high-level overview of our methodology. Input activations are mutable during inference, hence they must be scaled at runtime (Figure 2-1) before being deployed in compute memories. When ReLU activation functions are employed, as common in modern CNNs, they assume positive values, so they can be stored in unsigned format. Note that CNNs inputs are usually unsigned by definition, e.g., being RGB values of an image.

In order to avoid overflow during multiplication, input activation values are scaled in the range $[0, 1)$, as follows:

$$x_i^* = \frac{x_i}{X_{max}} \tag{4}$$

where X_{max} is largest activations value.[1] After scaling, input activations (Figure 2-2) can be represented in unsigned $Q0.N$ format (i.e., using no integer or sign bits, but only N fractional bits) without incurring overflows during quantization.

As mentioned in Section 3, in the case of convolutional layers input activations are then stored as fixed bitwidth, fixed-range numbers in the compute memory arrays, assuming the role of In-Memory Operands (IMOs). As shown in the next section, multiple bitwidths for IMOs can be supported by properly programming the carry chain. In our implementation, we consider 8-bit and 16-bit values per word, hence N can be either 8 or 16 in different layers.

Conversely, weights in convolutional layers are processed by all subarrays in parallel bit-by-bit as Broadcasted Operands (BOs). To this end, they are pre-scaled in the range $[-1, 1)$ and encoded in the $Q1.M$ format, i.e., as signed two's complement numbers with one integer bit. Being weight values constant during inference, their scaling can be done offline, before the CNN deployment. Moreover, since only one bit of BOs is processed at a time, no architectural constraint is present on M (the number of BOs fractional bits), which can hence be solely driven by accuracy considerations.

Scalar multiplications of weights and activations also reside in the memory array (Figure 2-3). These are signed values in the range $[-1, 1)$, being the product of an input in $Q0.N$ format and a weight in $Q1.M$ format. We encode them in $Q1.(N-1)$, using the same number of bits of the corresponding input activations by truncating and discarding the least significant M bits. This operation has a negligible impact on accuracy in practice, as shown in Section 8.

As opposed to multiplications, the accumulations of scalar products required to perform dot products do cause an increase in the data range. Nonetheless, the bitwidth increase is only logarithmically dependent on the number of accumulated values. In dot products, each scalar product is only accumulated once, hence the bitwidth increase is bounded by the memory array size. We show how such an increase is managed with low-complexity hardware in Section 6. After an output activation is computed (Figure 2-4), it is then post-scaled (Figure 2-5) according to Equation 5, producing the final output activations in floating-point representation (Figure 2-6):

$$y_k = y_k^* \cdot x_{max} \tag{5}$$

Our methodology also supports the execution in-/near-memory of fully connected layers in addition to the convolutional ones, described above. To this end, as detailed in [26], the role of weights and activations should be exchanged, because no weight re-use is present in fully connected layers (i.e., each weight is used only once), hampering parallelism if they are used as BOs. Thus, for these layers, weights should be employed as IMOs, with activations acting as BOs.

5 CNN OPTIMIZATION

5.1 Quantization methodology

Before deployment, CNNs are optimized to reduce their computational requirements. To this end, we devised a domain-specific and accuracy-driven quantization methodology which (a) leverages the runtime optimization opportunities offered by the compute memory architecture and (b) abides by a user-defined QoS constraint, expressed as a tolerable accuracy degradation with respect to a floating-point implementation. We will show in Section 8 that even very tight QoS constraints lead to very aggressive optimizations, with most data represented using just a handful of bits.

The optimization algorithm takes into account that

- The compute memories organization induces an asymmetry between BOs and IMOs. In particular, BOs, being processed bit-by-bit to control shift-add operations, can assume any bitwidth. Conversely, IMOs, being resident in memory, can only support a few choices, as their representation can only be manipulated by configuring the

[1]Symmetric activation functions, such as $tanh$, can be also supported. In this case, input activations of each layer are signed, and they are scaled in the $[-1, 1)$ range.

carry chain of the compute memory adder (see Section 6). Indeed, we consider 8-bit and 16-bit IMOs across our experiments. More aggressive bitwidth reductions of IMOs to just two or four bits are possible and may allow for higher degrees of parallelism. Nevertheless, supporting them would require additional logic for their implementation, directly impacting performance. Moreover, it would increase the optimization methodology complexity (and run-time) as more configurations have to be evaluated. In addition, we have observed that such low-precision IMOs have large detrimental effects on the accuracy of CNN models. Therefore, we are not considering them in the experimental exploration in Section 8.

- Compute memories support different BOs and IMOs bitwidths across layers. Such a high degree of flexibility opens the door to heterogeneous quantization strategies, but also leads to an exponential explosion of possible quantization choices with the number of layers in the CNN to be optimized.

To address the ensuing challenge, we devised a heuristic methodology that privileges optimization avenues leading to the highest runtime reductions. Given a compute memory architecture, a convolutional network model, and an initial quantized data representation, and assuming convolutional and fully connected layers implemented as a sequence of parallel shift and add operations, we can compute the total number of clock cycles for MAC operations in each layer as follows:

$$CLOCK\_CYCLES_i = \frac{width(BO_i) \cdot width(IMO_i) \cdot MAC_i}{MEMORY\_WORD_{size}} \tag{6}$$

where $width(BO_i)$ and $width(IMO_i)$ represent the bitwidth of BOs and IMOs in layer i, MAC_i is the total number of MAC operations executed in that layer, and $MEMORY\_WORD_{size}$ is the physical size of the memory words in the accelerator: in our design, it corresponds to 16 bits. Equation 6 assumes that one BO bit is processed at each clock cycle. We will show in Section 6 that this constraint can be relaxed with hardware optimizations, allowing to further speed-up computation. 16-bit IMOs and 8-bit BOs are considered in the baseline quantized models, since across all experiments this arrangement resulted in no accuracy degradation with respect to floating-point CNNs.

The equation above also allows computing the reduction of the number of clock cycles of MACs for a certain quantization choice. Reducing in one layer the BOs bitwidth by one bit reduces the number of shifts/adds required for each multiplication performed in that layer. Instead, halving the IMOs bitwidth in a layer also halves the number of MAC cycles in that layer, because two values can be stored in the same memory word. Hence, layer-wise quantization choices can be ordered from the one having the largest impact on clock cycles reduction (halving the IMOs of the largest CNN layer) to the one having the least impact (reducing the BOs width by one in the smallest layer).

We employ such an ordered list to drive our heuristic, as shown in Figure 3. The quantization that can lead to the highest runtime reduction, among those available (Figure 3-a), is selected at each optimization iteration (Figure 3-b). This approach allows us to select at each iteration which layer should be optimized, and, in that layer, whether to quantize more aggressively IMOs or BOs. Since bitwidth reductions usually negatively affect CNN accuracy, the model is then retrained for a limited number of epochs to compensate for a potential accuracy loss (Figure 3-c). Next, the accuracy of the newly optimized CNN model is compared with the user-defined accuracy threshold (Figure 3-d). If the desired output quality is reached, the optimization loop continues. Conversely, the applied optimization is first discarded and the previous bitwidth restored in the target layer (Figure 3-e). Additionally, the current action is removed from the potential candidate optimizations, so that it will not be considered again in future iterations (Figure 3-f). The heuristic terminates when no more optimization choices are available.

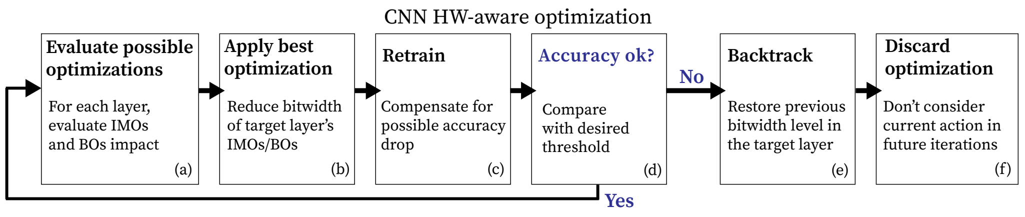

CNN HW-aware optimization
Evaluate possible optimizations
For each layer, evaluate IMOs and BOs impact
(a)
Apply best optimization
Reduce bitwidth of target layer's IMOs/BOs
(b)
Retrain
Compensate for possible accuracy drop
(c)
Accuracy ok?
Compare with desired threshold
(d)
No
Yes
Backtrack
Restore previous bitwidth level in the target layer
(e)
Discard optimization
Don't consider current action in future iterations
(f)

Fig. 3. Overview of the per-layer CNN quantization strategy. While further optimization actions are possible (a), the one having the largest impact on MAC cycles count reduction is selected and applied to the model (b). Then, a retraining phase compensates for the accuracy drop resulting from a bitwidth reduction of either IMOs or BOs (c). If the accuracy constraint is violated (d), the previous configuration is restored (e), and the current action is removed from the list of candidate optimizations (f).

5.2 Complexity analysis

The proposed quantization strategy loops over all the possible optimization actions that can be taken in each layer. As discussed in Section 5.1, two possible optimizations are possible in each layer: (a) reduction of IMOs bitwidth to 8 bits or (b) 1-bit reduction of BOs bitwidth. In both cases, the optimization step is followed by 20 retraining epochs, which constitute almost the entirety of the computation time in our methodology. From an optimization runtime perspective, the worst-case scenario is the one where all optimization steps are successfully applied to the model, so that the procedure continues for a high number of iterations. The IMOs can only be reduced to 8-bit values, and, in this context, have a much lower impact than BOs, whose bitwidth can assume any quantization level between 1-bit and 8-bit. Therefore, applying the proposed procedure to a model with N layers has a complexity in the order of $O(8N)$, ultimately being linear in the number of layers[2].

In contrast, an exhaustive exploration should evaluate all possible bitwidth combinations of IMOs and BOs of all layers. Therefore, although it will provide an optimal solution, it is unfeasible in practice, as its complexity is exponential in the number of layers. In particular, it has a complexity of $O((8 \cdot 2)^N)$, where 8 and 2 represent the number of possible BOs and IMOs bitwidths, respectively ($8 \cdot 2 = 16$ is the number of possible bitwidth combinations in *each* layer).

6 COMPUTE SRAMS DESIGN AND OPERATION

Starting point for our compute memory designs is a conventional SRAM array. We provide computational capabilities by interfacing a Processing Element (PE) to each of its subarrays, fully integrated into the memory elements (in the IMC case) or at the very close periphery (for the NMC design). Notably, the IMC approach uses the SRAM bit-lines to perform part of the computation, resulting in a more area-efficient implementation. On the other hand, NMC does not require modifications to the memory array. It is entirely based on standard cells design, hence requiring a much lower design effort.

Our designs perform multiply-accumulate operations based on repeated shifts and additions. Such a choice leads to low-area PEs, while allowing the execution of heterogeneously quantized CNNs, as defined by the per-layer optimization heuristic in Section 5. In particular, we assume that one or more bits of a Broadcasted Operand (BO) are used to control addition operations between an accumulator register and an In-Memory Operand (IMO). This strategy scales to BOs of any bitwidth. Different IMOs bitwidth can also be supported by configuring the carry chain for in-memory operations. We exploit this characteristic to allow the storage of either 1x16-bit values or 2x8-bit values for each memory word. As

[2]The optimization of any of the proposed benchmarks takes less than one day when performed on a Tesla V100 GPU.

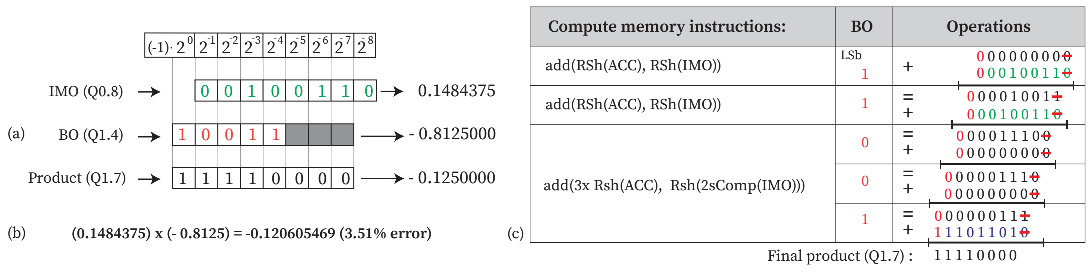

(-1)· 2^0 2^-1 2^-2 2^-3 2^-4 2^-5 2^-6 2^-7 2^-8
IMO (Q0.8) → 0 0 1 0 0 1 1 0 → 0.1484375
(a)
BO (Q1.4) → 1 0 0 1 1 → - 0.8125000
Product (Q1.7) → 1 1 1 1 0 0 0 0 → - 0.1250000
(b)
(0.1484375) x (- 0.8125) = -0.120605469 (3.51% error)
(c)
Compute memory instructions:
BO
Operations
add(RSh(ACC), RSh(IMO))
LSb
1
add(RSh(ACC), RSh(IMO))
1
add(3x Rsh(ACC), Rsh(2sComp(IMO)))
0
0
1
Final product (Q1.7) :
11110000

Fig. 4. (a-b) Multiplication between IMO in $Q0.8$, and BO in $Q1.4$ fixed-point formats, with result in $Q1.7$. (c) Corresponding compute memory instructions (additions, shifts, and two's complements).

discussed in Section 5, more fine-grained configurations (e.g., 4x4-bit or 8x2-bit values per word) are not implemented, as these low-precision representations significantly affect the evaluated benchmarks from both an accuracy and performance perspectives. Finally, BOs can be provided to multiple subarrays at the same time (hence the name), enabling the parallel computation of multiple MACs at the same time.

An example of the compute memory operations performed by a subarray is provided in Figure 4, which illustrates the multiplication between an unsigned IMO in Q0.8 format and a signed BO in Q1.4 format. The binary representation of the operands and the result are depicted on the left side of the figure (a,b), while the compute memory operations are shown on the right (c). From Figure 4-c, it can be noticed that the output and the IMOs have the same bitwidth, and that the output is approximated due to the truncation of one LSB for every shift right operation[3]. Moreover, the bitwise negation of the IMO operand is required when considering the most significant bit of the BO, in order to perform 2's complement arithmetic (this operation is then completed by setting up the adder carry-in as '1'). Finally, additions are only required when the BO bit is '1'. As in [27], we exploit this last observation by allowing up to three shift-in operations per clock cycle, so that BO patterns with leading '0's (such as "01" or "001") can be processed with a single compute memory operation.

In summary, the computation of multiplications requires support for addition, bitwise negation, and (arithmetic) right shifts. The next two subsections provide details on how these are implemented in an NMC and IMC approach, respectively. In the following, we also illustrate NMC and IMC mechanisms for overflow-free accumulation, enabling the memory-based computation of MACs, dot products operations, and, ultimately, convolutional and fully connected CNN layers.

6.1 NMC implementation

6.1.1 ***NMC architecture.*** A block scheme of the near-memory PE is shown in Figure 5. It embeds a 32-bit adder, as well as the logic for the bitwise negation and right shifts of an IMO value read from the memory. Moreover, it allows right-shifting by up to three bits of the partial product value (ACC). Three registers store a copy of an SRAM memory word (IMO reg), the partial product (ACC reg), and the MAC value (MAC reg) being computed.

6.1.2 ***NMC overflow avoidance.*** At runtime, when performing a multiplication the first adder operand is selected either as '0' (when processing the "000" BO pattern), the right-shifted IMO, or its 2's complement (when processing the BO most significant bit[4]). The second adder operand is the content of the ACC register, properly right-shifted by

[3]We analyze the effect of truncation errors on accuracy in Section 8.
[4]In this case, the carry-in of the adder is set to '1' to correctly perform the arithmetic negation of the IMO value.

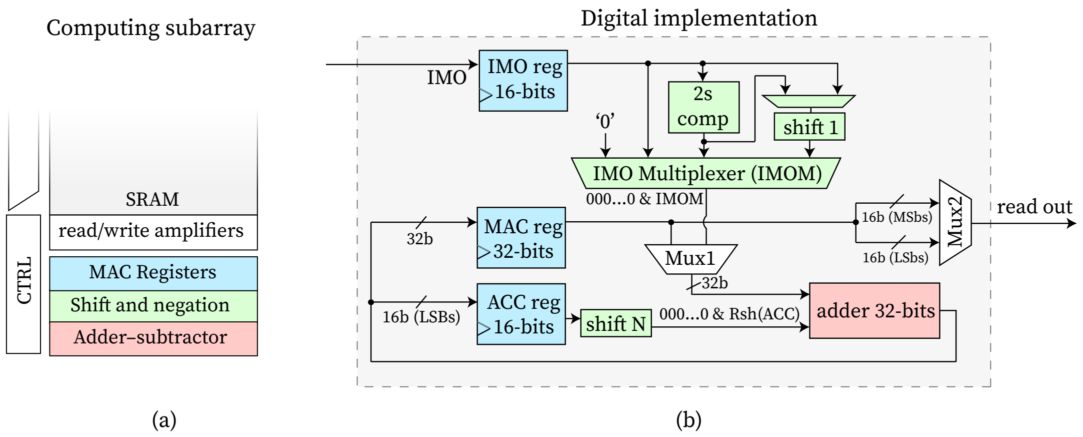


Fig. 5. (a) Generic basic PE composed of the IMO pre-processing circuit, registers for the multiply and accumulate operation, and the adder. (b) digital implementation of the NMC architecture.

one, two, or three bits. Once a multiply operation is finished, accumulation is performed as an addition between the ACC and MAC registers. Overflow is avoided in accumulations by sizing the MAC register to 32 bits. Finally, when all multiply-accumulate operations required for a dot product terminate, the two parts of the MAC register corresponding to the least significant and most significant 16 bits are read out.

6.1.3 ***NMC data level parallelism.*** Both 2x8-bit and 1x16-bit IMO operands are supported by configuring the connections in the carry chain of adders, so that in 2x8-bit mode carries do not propagate in-between subwords boundaries. Moreover, sign extension (instead of shifting) is performed between the 7-th and the 8-th bit in shifters.

6.2 IMC implementation

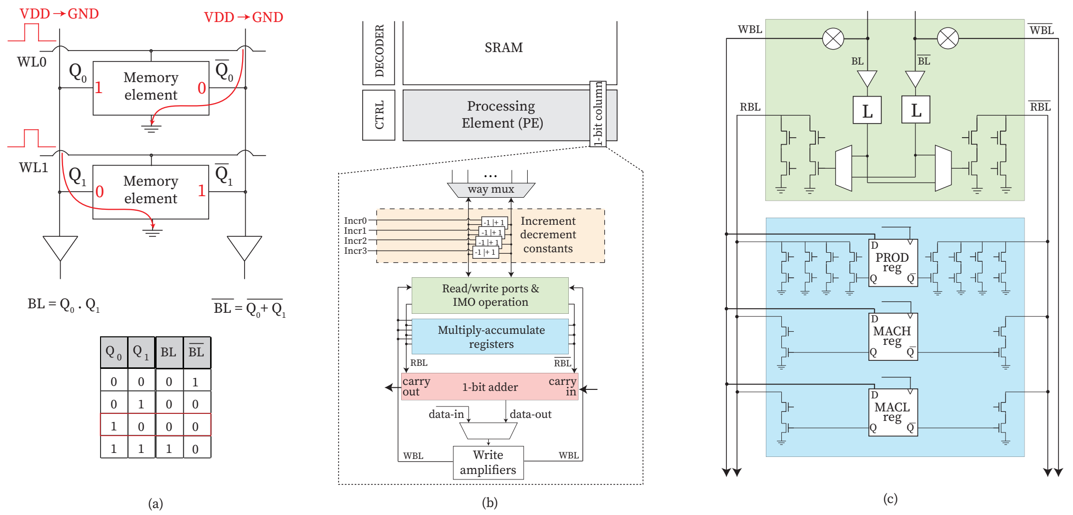
6.2.1 ***IMC architecture.*** The IMC design employs bit-lines computing, i.e., it exploits the discharge mechanism along bit-lines as the basis for deriving arithmetic computations. Such an approach can be applied whenever bit-lines (BL and $\overline{BL}$) are employed to access the bit value stored in a memory cell, regardless of the cell implementation. In the IMC design, we leverage this opportunity to allow computations between in-memory operands (stored in SRAM cells) and dedicated registers containing partial products and accumulation values (employing flip-flops).

In standard (i.e., non-IMC) accesses, BL and $\overline{BL}$ are pre-charged to V_{dd}. If the stored value is '0', the BL discharges to ground, while $\overline{BL}$ stays at V_{dd}. The opposite happens if a logical '1' is stored. The key insight of bit-line computing is that, if *two* word-line signals corresponding to different rows are activated concurrently, for each bit position, two discharge paths are presented, as depicted in Figure 6-a. Hence, BL and $\overline{BL}$ carry the binary AND and NOR signals of the corresponding bits of two different words, respectively.

The circuitry for a 1-bit column[5] of the IMC architecture exploiting bit-line computing is provided in Figure 6-b and further detailed in Figure 6-c. Similarly to the NMC design, the IMC architecture uses registers to store the ACC and MAC values. However, since all words must share the same bitwidth to perform bit-line computing, the MAC register is split into two parts hosting the most significant bits (MACH) and the least significant ones (MACL), respectively.

When first read, IMOs are latched, so that repetitive accesses to the same memory words (and the associated energy cost) when performing multiplications are avoided. By using latches (rather than flip-flops) the fetching of an in-memory

[5]We herein focus on a bit-column for clarity. Note that, upon access, all bits in a word are activated by a word-line.



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

Fig. 6. (a) Bit-line computing paradigm between two memory elements and its bitwise logic operations AND and NOR. (b) Block diagram of the IMC architecture, focusing on a 1-bit column of the PE. (c) Detailed circuit of the read ports and registers. $RBL\backslash\overline{RBL}$, $WBL\backslash\overline{WBL}$ refer to the read and write path of the bit-lines, respectively.

operand and the first shift-add operation can be performed in the same clock cycle, speeding up execution with respect to the NMC design. Since results of in-memory operations are stored in registers (as opposed to SRAM cells), write-backs to the SRAM memory array are avoided. Therefore, the assertion of the write bit-lines (WBL) can be performed in parallel with the pre-charging of read bit-lines (RBLs), since the read/write paths of flip-flops are separated, differently from SRAMs. This results in a 2× acceleration of multiply operation compared to state-of-the-art bit-line computing architectures [31] [26]. On the other end, in the IMC case, two cycles are required for accumulation (to update MACH and MACL, respectively). Ultimately, all the above considerations result in equal performance for the IMC and NMC designs in terms of clock cycles.

The IMC architecture supports arithmetic shifts, negations, and additions to implement MACs, dot products, and the in-memory computation of CNN layers. Bitwise negation of IMOs is performed by modifying read ports level, embedding two multiplexers that, inside the read port, may exchange the BL and $\overline{BL}$ signals (Figure 6-c). This operation, in conjunction with the assertion of the carry-in of the LSb in the embedded adder, performs the arithmetic negation of an IMO in 2's complement representation. Shifts on IMOs and on the ACC register are also supported by customizing the memory array read-out circuitry. To this end, we provide connections from the outputs of each bit column to the bit-lines of the neighboring ones using additional read ports, realized with two NMOS in series. Finally, additions are performed by deriving, for each bit-column, the sum and carry-out signals from the ANDs and NORs present on BL and $\overline{BL}$, and connecting carry-ins and carry-outs from all bit-columns.

6.2.2 ***IMC overflow avoidance.***

During multiplications, in-memory shift-add operations are employed as described in Section 6, relying on fixed-point representation to avoid overflow. Special care must instead be taken when performing accumulation. In this case, the result of a product stored in the ACC register is first added to the content of the MACL register using the in-memory adder. Note that this operation may induce positive or negative overflow. As depicted in Figure 7, the type of overflow is dictated by the carry-out and sum bits of the input MSbs: when these are equal, no overflow occurs, while, if they differ, a positive ("01") or negative ("10") overflow occurs.

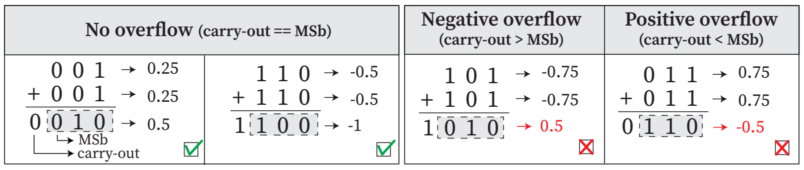


Fig. 7. Overflows from additions of two values represented in Q1.2 format.

Table 1. Baseline floating-point accuracy (Top-1 and Top-5) and model complexity (model size and computing requirements) of the considered CNNs evaluated on the CIFAR100 dataset.

Benchmark	Top-1 Accuracy (%)	Top-5 Accuracy (%)	Model size [MB]	GFLOPs
AlexNet	62.73	87.02	14.84	1.04
VGG16	60.39	83.37	14.62	0.29
ResNext	73.07	91.52	20.14	1.11
GoogLeNet	72.11	91.35	5.42	0.34
ResNet-8	59.53	85.27	5.88	0.19
MobileNet-v2	47.19	75.12	3.15	0.32

We use those bits to govern the state of the MACH register. In case of no overflow, its value is retained. Instead, in case of overflows, the sum and carry-out signals of the MSb of MACL registers are used to index two constants, corresponding to the values '-1' and '+1'. An in-memory addition is then triggered between the selected constant and MACH, with the result being written back to MACH.

6.2.3 ***IMC data level parallelism.***

As in the NMC design, the IMC architecture can also support word-level parallelism i.e., storing multiple values in different bit-fields of a memory word. In our implementation, we show this capability by allowing either 1x16-bit or 2x8-bit word formats. In the latter case, the separation among sub-words during right-shift operations is supported by performing sign extensions instead of shifting in the 8-th bit-column. Moreover, the carry-in signal of the 8-th bit can be either connected to that of the 7-th bit (in 1x16-bit mode) or be asserted/de-asserted for addition/subtraction in 2x8-bit mode.

Finally, when updating the MACH value to account for positive/negative overflows of MACL, we provide a mechanism that allows managing two sub-words separately. First, we employ separate read enables for the increment/decrement constants (Figure 6) for each subword. These constants are only asserted when a MACH subword increment/decrement must be performed upon an overflow in MACL. Then, we provide four constants, encoding the values $\{(+1, +1), (+1, -1), (-1, +1), (-1, -1)\}$ to manage all positive/negative MACH update combinations across subwords.

7 EXPERIMENTAL SETUP

7.1 Baselines

7.1.1 ***Quantization methodology.***

We briefly compare in Section 8.2 the performance of our quantization approach with respect to the accuracy-driven strategy in [21], in terms of number of achieved MAC reduction at iso-accuracy. Moreover, in Section 8.3.2 we adopt as a baseline accuracy level the one achieved using "fake quantization" [11]. Notice that this is an upper bound with respect to our method, as fake quantization does not enforce constrained data ranges in intermediate results, such as product and accumulation values.

7.1.2 ***Overflow-free IMC and NMC architectures***.

To give a detailed view of the performance and resource requirements of our design, in Section 8.2 we compare achieved different runtime and accuracy figures with respect to compute memory solutions that do not feature specialized support for overflow management, as is the case in the state of the art [31][18]. To this end, we considered two baseline strategies. In the first one, as in [41][21], overflow is managed (when it occurs) by saturating the result of an operation to the maximum representable negative or positive values. We refer to this baseline as *Saturation* in our discussion.

The second baseline quantizes inputs to a degree that guarantees that overflows do not happen, even if the accumulator value is stored in a single memory word, as assumed in [31] and [18]. Note that such baseline strategy requires the use of many memory bits solely for the sign-extension of inputs, instead of using them to store more precise data values or to enable data level parallelism, as we do in our proposed approach. Indeed, considering as an example 256 rows and 1x16-bit IMOs, this baseline strategy would restrict the IMOs useful bitwidth (i.e. apart from sign extensions) to only 8 bits. Additionally, it entirely disallows a 2x8-bit IMOs configuration, as the most significant bits of IMOs are required to store sign information. We refer to this baseline as *8-bit IMOs*, as all IMOs are quantized as 8-bit values, but sign-extended to 16 bits.

7.2 Benchmarks

We gathered results of accuracy and runtime performance on a collection of CNN benchmarks in the field of edge AI, namely AlexNet [16], ResNet8 [7], VGG16 [32], GoogLeNet [35], ResNext [40], and MobileNet-v2 [9]. All applications were evaluated on the CIFAR-100 dataset [15]. As a consequence, minor adjustments in the first layers were needed to adapt these models to the image sizes of the target dataset (i.e., 32x32 RGB inputs). The proposed benchmarks exhibit different degrees of complexity, and therefore are good candidates to evaluate our methodology across a vast range of edge AI applications. They differ in the number of parameters, size, depth, and type of connections of the layers. For example, AlexNet uses very large convolutional layers and is the only architecture not using batch normalization layers. VGG16 has a similar, but much deeper structure (i.e., it has more layers than AlexNet) and includes batch normalization layers after every convolution. ResNet-8 and ResNext make use of residual/skip connections, with the latter model being more complex and deeper. GoogLeNet is composed of a series of inception blocks, that calculate the channels of the corresponding output features using different convolutional layers. Finally, MobileNet-v2 is specifically designed for edge applications and includes separable convolutions, a combination of depth-wise and point-wise convolutional layers, which ultimately reduce the number of MACs and required parameters. Table 1 reports the achieved Top-1 and Top-5 accuracies, as well as the size and computational complexity of the presented benchmarks.

7.3 CNN training environment

CNN models were trained and quantized in PyTorch. First, the models were trained using floating-point precision until convergence. The BOs and IMOs were then quantized using 8-bit and 16-bit representations, respectively (a quantization scheme usually referred to as the 8/16 quantization level). As quantization may impact CNN accuracy, an additional 20 retraining epochs were run, forcing BOs and IMOs to assume quantized values. The retraining phase allows the 8/16 quantized benchmarks to recover in all cases from the accuracy drop with respect to floating-point baselines. Finally, we optimized our benchmarks using the heterogeneous quantization strategy per layer described in Section 5, with an accuracy degradation constraint of 1% with respect to the 8/16 quantized models from the baseline.

7.4 Accuracy and runtime exploration

7.4.1 ***C++ inference solver for accuracy evaluations.*** The presented accuracy numbers were evaluated via a custom inference solver written in C++. The inference solver can execute convolutional, fully connected, pooling, and normalization layers. CNN models are defined by connecting these layers, effectively implementing different CNN architectures. The specific characteristics of each layer of a CNN (i.e., input size, number of input/output channels, kernel size, among others) are defined in a header file that represents the CNN model.

The solver executes real fixed-point arithmetic, using the actual operations performed by the compute memories, as described in Figure 4. To this end, fake quantization floating-point weights generated in PyTorch are converted into the corresponding integer representations. Then, each multiplication performed in a convolutional or fully connected layer is replaced by custom functions that perform shifts, additions, and two's complement. Hence, the solver performs the same arithmetic operations as would be done by the compute memories, allowing to explore the ensuing accuracy under different scenarios, such as our proposed implementation, as well as the baselines *Saturation* and *8-bit IMOs*.

7.4.2 ***Runtime evaluations.*** Similarly to [26], the execution and mapping of the convolutional and fully connected layers in compute arrays were carried out layer-by-layer. To evaluate the cycle count of inference, we used the CNN2BLADE cycle-accurate simulator [6]. The simulator tiles the input activations for each layer to minimize data transfers and maximize parallelism, considering architectural constraints (e.g. the size of memory subarrays and their number) and application characteristics (such as the bitwidth of IMOs and BOs in each layer). Moreover, the simulator also computes the number of clocks required to aggregate partial convolution results, when their computation is distributed across different subarrays in order to maximize parallelism.

7.5 Implementation

We designed the SRAM array using 28nm TSMC CMOS technology, implementing high-density ($0.127\mu m^2$) memories using 6T SRAM cells. As a test vehicle for both IMC and NMC, we considered memories composed of a varying number of 2KB subarrays, each organized as 1024 words of 16 bits each. We bit-interleaved four words in each memory row, so that the implemented array circuit presents 64 BLs and 256 WLs. This configuration allows up to 2.2GHz read and write memory operations. We use this timing constraint to design and optimize IMC and NMC solutions. The IMC architecture was implemented as a full custom design. The IMC PE's energy and propagation times were characterized considering the parasitic equivalent given by RC networks. The circuit was simulated using the hSpice simulator. On the other hand, the NMC architecture was implemented as a semi-custom design. Its behavior was described in RTL and synthesized using standard digital cells from the TSMC 28nm PDK.

The energy characterization considered the average values derived from the simulation of the different MAC operations with different input bit patterns. In the IMC case, the patterns were employed as inputs to hSpice simulations. For the NMC implementation, they were instead used to extract switching activities with RTL simulation, which are in turn employed to derive the accurate NMC power model.

[6]https://github.com/esl-epfl/CNN2BLADE

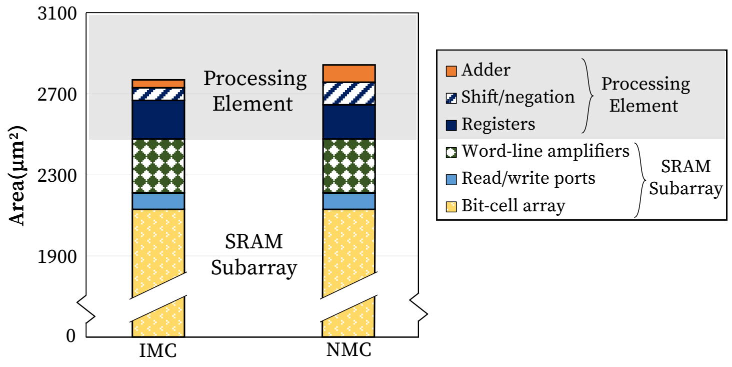


Fig. 8. SRAM subarray, IMC, and NMC Processing Element (PE) area breakdown. Area values are expressed in μm^2.

Table 2. SRAM subarray (256WL, 64BL), IMC, and NMC Processing Element (PE) energy breakdown.

SRAM Subarray	**Energy (fJ)**	**IMC PE**	**Energy (fJ)**	**NMC PE**	**Energy (fJ)**
Read	491.6	Registers	155.8	Registers	127.3
Write	363.6	16-bits adder	31.9	32-bits adder	64.2
Leakage	88.9	Shift/negation	51.0	Shift/negation	83.7
		Total	238.6	**Total**	275.2

8 EXPERIMENTAL RESULTS

8.1 Circuit Analysis

8.1.1 ***Area***. Figure 8 presents as stacked bars the area breakdown of the IMC (left bar) and NMC (right bar). The bottom part of the bars reports the area of an ordinary SRAM subarray (i.e., a memory subarray with no computing capabilities). Their top part instead shows the area of the IMC and NMC PEs. The main contributor to the area is the SRAM bit-cells, which represent 77% of the IMC area and 75% of the NMC area. The overhead of implementing the PEs as IMC or NMC, with respect to an ordinary SRAM subarray, represents only 10.5% and 12.9%, in the two cases.

The IMC implementation presents the most compact PE. The reason is two-fold: first, the IMC design shares the read/write ports (3% of the area) with the SRAM subarray to perform in-situ shift/negation operations instead of using logic gates, such as the NMC. Second, using a 16-bit adder based on the AND and NOR logic operation of bit-line computing implies only 1.5% area overhead, whereas the 32-bit adder of the NMC represents 3.3%. However, different from the shift/negation and adder, the registers at the IMC architecture represent more area (7.15%) compared to the NMC architecture (6.39%).

8.1.2 ***Energy***. Table 2 shows the energy comparison between the SRAM array, the IMC, and NMC PEs. The SRAM subarray requires 491.6 fJ for a 16-bit read operation and 363.6 fJ for a 16-bit write operation, while consuming 88.9 fJ as static energy (leakage). The IMC PE consumes a total of 238.6 fJ to compute a shift-add operation, which represents a 50% energy reduction compared to IMC operations performed with data accessed directly from the SRAM array, such as [26]. On the other hand, the NMC PE consumes 16% more energy than the IMC PE. It can be noticed that the NMC adder and shift/negation logic consumes more than 2× energy compared to its IMC peers. The reason is that the IMC PE extracts logic operations passively from the pre-charged bit-lines, reducing its energy consumption. However, IMC

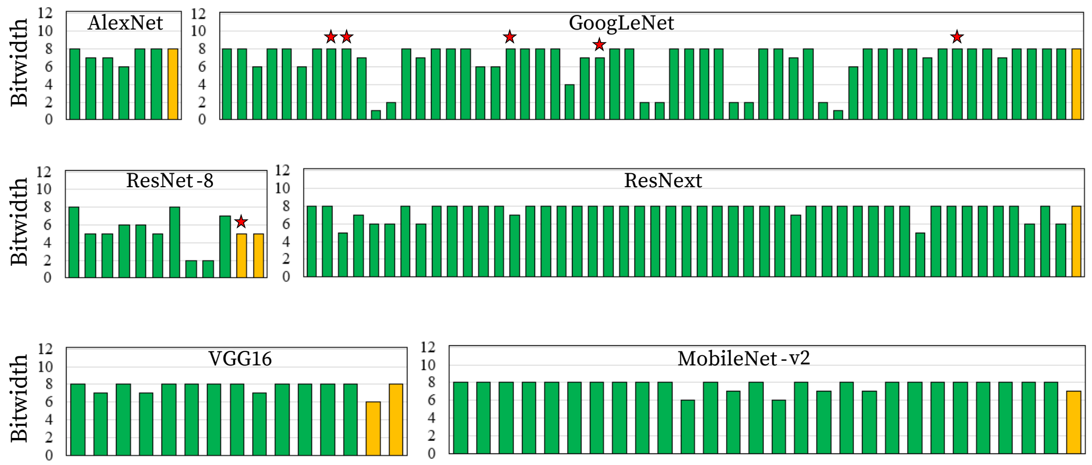


Fig. 9. Per-layer bitwidth of BOs (green bars for convolutional layers and yellow bars for fully connected ones) in the optimized benchmarks. IMOs are assumed to be reduced from the baseline 16-bit to 8-bit words, except for those in layers marked with a red star.

registers consume 25% more energy compared to NMC because the IMC solution requires more registers to manage the parallelization of write-back/pre-charge, as explained in Section 6.

8.2 Per-layer CNN Optimization

The selected benchmarks are optimized using the heterogeneous quantization strategy discussed in Section 5. Figure 9 illustrates the bitwidth of BOs and IMOs in our optimized CNN models. In particular, green and yellow bars show the bitwidth of BOs in convolutional and fully connected layers, respectively. Our heterogeneous quantization strategy reduces IMOs to 8 bits in all layers, except for those marked with a ★. The achieved results indicate that, in general, IMOs are the preferred target for optimization, as their bitwidth reduction impacts the total number of MAC cycles significantly more than BOs. Nevertheless, they may also have a larger impact on accuracy, and, in some CNN models, certain layers must keep 16-bit IMOs to limit their accuracy drop to less than 1%. On average, our optimization strategy reduces the number of shift-add cycles by 55.46%, thus accelerating the inference runtime and reducing the energy. Compared to the optimization strategy presented in [21], and considering only the common benchmarks evaluated, the proposed solution reduces the number of MAC cycles by 13% more, on average.

8.3 Comparative Architectural evaluation

8.3.1 ***Effect of partial products truncation***. The in-memory implementation of approximate multiplications relies on the iterative truncation of partial products, represented in $Q1.X$ fixed-point format. To measure the magnitude of introduced errors in the output products, we executed shift-add multiplications, as implemented in our IMC design, over all possible input combinations (IMO, BO). Our results indicate that the effect of such approximations is negligible, as it introduces a Mean Relative Error (MRE) of only 1.6% in the output results. Most importantly, such MRE causes an insignificant average drop of just 0.11% and 0.05% in the Top-1 and Top-5 accuracies, respectively, in our evaluated benchmarks.

8.3.2 ***Accuracy-performance trade-off***. Figure 10 compares our approach with the two alternative overflow handling techniques discussed in Section 7. The analysis presented in this section focuses on the performance of the inference

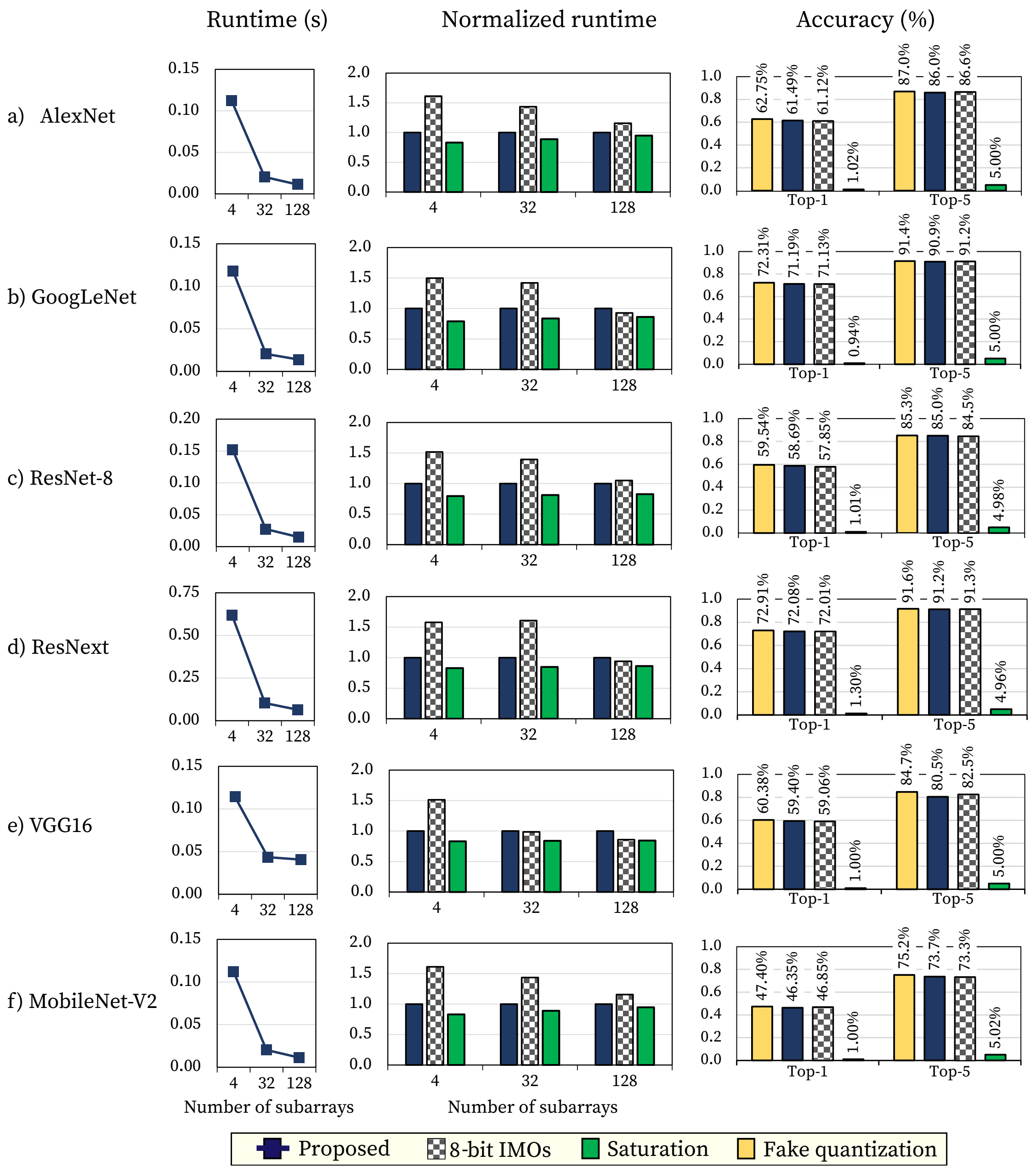


Fig. 10. Absolute inference runtime of the heterogeneously quantized CNN benchmarks executed in our proposed overflow-free architecture for different subarray arrangements (left). The runtime of *8-bit IMOs* and *Saturation*, normalized to our proposed solution (center). The accuracy achieved in the three different architectural designs and, for reference, with fake quantization (right).

and the accuracy achieved. The plots on the figure's left show the absolute runtime, expressed in seconds, of the heterogeneously quantized benchmarks executed in the proposed overflow-free architecture. The barplots at the center instead show the inference runtime of the three different architectural designs normalized with respect to that of our proposal. We report the runtime required for a single inference in both graphs, considering an increasing number of memory arrays (i.e., 4, 32, and 128 subarrays). In this regard, our results assume that a unique BO is broadcasted to all the arrays at each IMC operation.Finally, the barplots on the right of Figure 10 depict the Top-1 and the Top-5 accuracies achieved by the fake quantization strategy and the three different approaches.

The Top-1 accuracy achieved by the proposed solution is around 1% lower than fake quantization. Such a (small) difference is caused by the truncation of partial products required in our scenario, which are not accounted for in fake quantization, as discussed in Section 7. Nonetheless, fake quantization can't be readily implemented in compute memories, because it does not account for limited data ranges.

The results presented refer to both the IMC and the NMC implementations, as the two designs require the same number of clock cycles for executing MAC operations. The results reflect that both the *Saturation* and *8-bit IMOs* baselines employ only one cycle for both accumulations (since they only employ one MAC register), while our solution requires two (to update MACL and MACH).

The plots on the left indicate that our approach scales nicely when increasing the number of subarrays. In fact, in comparison with a 4-subarray design, we measure, on average, inference speed-ups of 5.41× and 9.92× in 32- and 128-subarray architectures, respectively. Nevertheless, we observe diminishing performance gains when moving from 32 to 128 subarrays, as the convolutional or fully connected layers are not fully exploiting the data parallelism offered by the hardware. Therefore, for edge AI applications where the number of convolutional filters is usually bounded to a few hundred, a large number of subarrays results in memory under-utilization.

When comparing our solution with the *8-bit IMOs* baseline, we observe speed-ups of 60%, on average. Performance gains are mainly due to the data parallelism possibilities enabled in our solution, which are instead prevented in the *8-bit IMOs* alternative, as the 8 MSbs of each word must not be used, to ensure overflow-free accumulations. This also translates into a non-optimal use of memory for the *8-bit IMOs* implementation, as 50% of memory words cannot be used for computation. Nevertheless, the *8-bit IMOs* approach results in slightly faster executions in 128-subarray implementations, as the inability of our proposal to fully exploit data parallelism cannot balance the faster MAC executions of *8-bit IMOs* (i.e., one clock cycle vs. two clock cycles per-MAC of our implementation). As overflow is prevented in both solutions, the only (negligible) impact on accuracy is due to the shift-add implementation of multiply instructions.

Compared to the *Saturation* solution, our strategy results in a 12% increase in runtime. The reason is that we assume that saturation operations have no impact on runtime. This assumption puts the *Saturation* baseline at an advantage because, in our proposed implementation, one additional IMC operation is required to update the value of the *MACH* register after each MAC operation. However, saturating the accumulator has a very adverse effect on accuracy, which drops dramatically in all cases (that is, ∼1% Top-1 and ∼5% Top-5 accuracies across all benchmarks, indicating random predictions in the 100-class classification problem of CIFAR-100). The number of saturated elements for each inference depends on multiple factors, including the CNN structure and the specific input pattern, ranging in our experiments from just 2% to more than 90%. On average, we measure 37% of saturated activations across the evaluated benchmarks, thus motivating the very low accuracy of this baseline.

9 CONCLUSIONS

In this work, we have presented a hardware/software co-design methodology that combines a hardware-aware heterogeneous quantization methodology and inter-layer operands scaling with lightweight hardware support to enable overflow-free in-/near-memory computation of dot products. Experimental results show that the timing overhead of operand scaling operations scales linearly with the data size and accounts for less than 0.1% of the inference runtime. We have evaluated the proposed overflow-free methodology of the resulting IMC and NMC approaches, measuring area overheads of just 10.5% and 12.9%, respectively, compared to a 2KB SRAM array without computing capabilities. Focusing on the acceleration of convolutional and fully connected layers, we have tested our implementations on a collection of edge AI CNN models, observing minimal effects on accuracy due to numerical truncations. We showcased that alternative compute memory solutions, not employing our overflow handling approach, result in either much higher runtime performance or unacceptable QoS degradations, while both downsides are avoided in our IMC/NMC overflow-aware designs.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Stephan Patrick Baller, Anshul Jindal, Mohak Chadha, and Michael Gerndt. 2021. DeepEdgeBench: Benchmarking deep neural networks on edge devices. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 20–30.

[2] Geoffrey W Burr, Robert M Shelby, Abu Sebastian, Sangbum Kim, Seyoung Kim, Severin Sidler, Kumar Virwani, Masatoshi Ishii, Pritish Narayanan, Alessandro Fumarola, et al. 2017. Neuromorphic computing using non-volatile memory. *Advances in Physics: X* 2, 1 (2017), 89–124.

[3] Kyubaik Choi and Gerald E. Sobelman. 2022. An Efficient CNN Accelerator for Low-Cost Edge Systems. *ACM Trans. Embed. Comput. Syst.* 21, 4, Article 44 (aug 2022), 20 pages.

[4] Claudionor N Coelho Jr, Aki Kuusela, Shan Li, Hao Zhuang, Jennifer Ngadiuba, Thea Klaeboe Aarrestad, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers. 2021. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence* 3, 8 (2021), 675–686.

[5] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 283–295.

[6] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2022. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access* 10 (2022), 52565–52608.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[8] Mark Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14.

[9] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[10] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. *Advances in neural information processing systems* 29 (2016).

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.

[12] MM Kamruzzaman. 2021. New opportunities, challenges, and applications of edge-AI for connected healthcare in smart cities. In *2021 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 1–6.

[13] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. I-BERT: Integer-only BERT Quantization. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.).

PMLR, 5506–5518.

[14] Joshua Klein, Irem Boybat, Yasir Qureshi, Martino Dazzi, Alexandre Levisse, Giovanni Ansaloni, Marina Zapater, Abu Sebastian, and David Atienza. 2022. ALPINE: Analog In-Memory Acceleration with Tight Processor Integration for Deep Learning. *IEEE Trans. Comput.* (2022).

[15] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2017. Imagenet classification with deep convolutional neural networks. *Commun. ACM* 60, 6 (2017), 84–90.

[17] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, et al. 2021. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2 tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 350–352.

[18] Kyeongho Lee, Jinho Jeong, Sungsoo Cheon, Woong Choi, and Jongsun Park. 2020. Bit parallel 6T SRAM in-memory computing with reconfigurable bit-precision. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[19] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. 2022. A survey on the convergence of edge computing and AI for UAVs: Opportunities and challenges. *IEEE Internet of Things Journal* (2022).

[20] Flavio Ponzina, Giovanni Ansaloni, Miguel Peón-Quirós, and David Atienza. 2022. Using Algorithmic Transformations and Sensitivity Analysis to Unleash Approximations in CNNs at the Edge. *Micromachines* 13, 7 (2022), 1143.

[21] Flavio Ponzina, Marco Rios, Giovanni Ansaloni, Alexandre Levisse, and David Atienza. 2021. A flexible in-memory computing architecture for heterogeneously quantized CNNs. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 164–169.

[22] Akshay Krishna Ramanathan, Gurpreet S Kalsi, Srivatsa Srinivasa, Tarun Makesh Chandran, Kamlesh R Pillai, Om J Omer, Vijaykrishnan Narayanan, and Sreenivas Subramoney. 2020. Look-Up Table based Energy Efficient Processing in Cache Support for Neural Network Acceleration. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 88–101.

[23] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: A framework for quantifying the resilience of deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[24] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2021. AI accelerator survey and trends. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[25] Marco Rios, Flavio Ponzina, Giovanni Ansaloni, Alexandre Levisse, and David Atienza. 2022. Error resilient in-memory computing architecture for cnn inference on the edge. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 249–254.

[26] M. Rios, F. Ponzina, A. Levisse, G. Ansaloni, and D. Atienza. 2023. Bit-Line Computing for CNN Accelerators Co-Design in Edge AI Inference. *IEEE Transactions on Emerging Topics in Computing* 01 (2023), 1–14.

[27] Marco Rios, William Simon, Alexandre Levisse, Marina Zapater, and David Atienza. 2019. An associativity-agnostic in-cache computing architecture optimized for multiplication. In *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 34–39.

[28] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. 2020. Memory devices and applications for in-memory computing. *Nature nanotechnology* 15, 7 (2020), 529–544.

[29] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 273–287.

[30] Patrick Siegl, Rainer Buchty, and Mladen Berekovic. 2016. Data-centric computing frontiers: A survey on processing-in-memory. In *Proceedings of the Second International Symposium on Memory Systems*. 295–308.

[31] William Andrew Simon, Yasir Mahmood Qureshi, Marco Rios, Alexandre Levisse, Marina Zapater, and David Atienza. 2020. BLADE: An in-cache computing architecture for edge devices. *IEEE Trans. Comput.* 69, 9 (2020), 1349–1363.

[32] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[33] Srivatsa Srinivasa, Akshay Krishna Ramanathan, Jainaveen Sundaram, Dileep Kurian, Srinivasan Gopal, Nilesh Jain, Anuradha Srinivasan, Ravi Iyer, Vijaykrishnan Narayanan, and Tanay Karnik. 2021. Trends and Opportunities for SRAM Based In-Memory and Near-Memory Computation. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 547–552.

[34] Sreenivas R Sukumar, Jacob A Balma, Cong Xu, and Sergey Serebryakov. 2021. Survival of the Fittest Amidst the Cambrian Explosion of Processor Architectures for Artificial Intelligence. In *2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC)*. IEEE, 34–43.

[35] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[36] Anton V Trusov, Elena E Limonova, Dmitry P Nikolaev, and Vladimir V Arlazarov. 2021. p-im2col: Simple yet efficient convolution algorithm with flexibly controlled memory overhead. *IEEE Access* 9 (2021), 168162–168184.

[37] Naveen Verma, Hongyang Jia, Hossein Valavi, Yinqi Tang, Murat Ozatay, Lung-Yen Chen, Bonan Zhang, and Peter Deaville. 2019. In-Memory Computing: Advances and Prospects. *IEEE Solid-State Circuits Magazine* 11, 3 (2019), 43–55.

[38] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2016. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4820–4828.

[39] Qiangfei Xia and J Joshua Yang. 2019. Memristive crossbar arrays for brain-inspired computing. *Nature materials* 18, 4 (2019), 309–323.

[40] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1492–1500.

[41] Chun-Chieh Yang, Yi-Ru Chen, Hui-Hsin Liao, Yuan-Ming Chang, and Jenq-Kuen Lee. 2023. Auto-Tuning Fixed-Point Precision with TVM on RISC-V Packed SIMD Extension. *ACM Trans. Des. Autom. Electron. Syst.* 28, 3 (2023).

[42] Taegeun Yoo, Hyunjoon Kim, Qian Chen, Tony Tae-Hyoung Kim, and Bongjin Kim. 2019. A Logic Compatible 4T Dual Embedded DRAM Array for In-Memory Computation of Deep Neural Networks. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6.

[43] Yiqun Zhang, Li Xu, Qing Dong, Jingcheng Wang, David Blaauw, and Dennis Sylvester. 2018. Recryptor: A reconfigurable cryptographic cortex-M0 processor with in-memory and near-memory computing for IoT security. *IEEE Journal of Solid-State Circuits* 53, 4 (2018), 995–1005.

[44] Zhixiao Zhang, Jia-Jing Chen, Xin Si, Yung-Ning Tu, Jian-Wei Su, Wei-Hsing Huang, Jing-Hong Wang, Wei-Chen Wei, Yen-Cheng Chiu, Je-Min Hong, et al. 2019. A 55nm 1-to-8 bit Configurable 6T SRAM based Computing-in-Memory Unit-Macro for CNN-based AI Edge Processors. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 217–218.

Received March 23, 2023; revised June 2, 2023; accepted June 30, 2023