EPFL

Thèse n° 10318

Hardware-Software co-design Methodologies for Edge AI Optimization

Présentée le 25 septembre 2023

Faculté des sciences et techniques de l'ingénieur
Laboratoire des systèmes embarqués
Programme doctoral en génie électrique

pour l'obtention du grade de Docteur ès Sciences

par

Flavio PONZINA

Acceptée sur proposition du jury

Prof. A. M. Alahi, président du jury
Prof. D. Atienza Alonso, directeur de thèse
Prof. T. Simunic Rosing, rapporteuse
Prof. L. Pozzi, rapporteuse
Prof. A. Burg, rapporteur

École polytechnique fédérale de Lausanne

2023

Acknowledgements

EXTREMELY hard is to include in this section all the people who had an impact on my life and allowed me to reach this achievement. Starting from the beginning, the first mention goes to my *family*, who intensively supervised my studies during the very first years of school, always stressing the importance of school, respect, and commitment.

I am really grateful I had Prof. *Massimo Castagno* for three years at high school. In addition to his technical classes on computer science, he teaches his students responsibility and method. I believe his words significantly contribute to my growth, as a professional and, even more, as a person. During the same years, I met a real friend, *Davide Saggese*, who motivated me to improve continuously and with whom I spent some of my best days.

At Politecnico di Torino, I had the chance to be in contact with so many good professors and researchers. I want to recognize Prof. *Andrea Calimera* and Dr. *Valentino Peluso* for the opportunity they gave me to work on cutting-edge research on Artificial Intelligence for my master's thesis. Valentino spent so much time teaching me and supervising my work, and I am really thankful for what he has done. Let me also recognize his patience in making me understand topics that took me time to (hopefully) master.

The Embedded Systems Laboratory (ESL) of EPFL welcomed me in 2019 and allowed me to broaden my horizons in these last years. I had the honor to join the lab to pursue a Ph.D., which is something I had never considered at an early age, and that I usually thought of as something much bigger than me

(even recently when seeing the great work of other students in the lab). Here, I met people from all over the world, who definitely changed the way I look at my future. We are one of the largest labs in EPFL, so citing all of them would result in an incredibly long list. I want to thank *Denisa, Davide, Giovanni, Miguel, José, and Tomás*, post-docs of the lab who taught me a lot. I also met really astonishing colleagues like *Elisabetta, Darong, and Lara*, who I really admire for their commitment and passion for what they do.

I also want to thank the commission who reviewed my thesis and helped me improve the clarity of its content. Namely, I thank Professors *Pozzi, Rosing, Alahi*, and *Burg* for their time spent reading and evaluating this document. Moreover, I must also thank them for accepting to be part of my Ph.D. commission and especially for being available to attend my private defense on a late Friday afternoon!

Finally, I must conclude these acknowledgments by mentioning the person who really had the most considerable impact on my work in these years: Professor *David Atienza*, my thesis supervisor. Always available and supportive when I needed help. Most importantly, he believed in me and, by accepting me as a Ph.D. student in his laboratory, he gave me the most valuable opportunity I had in my life. Not only that, but he is still assisting me in my future career, supporting among other things, my application for a postdoctoral position abroad.

Thanks, David.

Lausanne, September 9, 2023 Flavio Ponzina

Abstract

ARTIFICIAL Intelligence (AI) is revolutionizing a vast range of industrial and scientific applications due to its several advantages, which include self-learning capabilities, extraction of intrigued hidden patterns from input data, and flexibility. While the cloud-based computing paradigm has been a baseline approach for AI inferences in the past years, recent technology advancements and AI optimization methods advocate and support a shift toward an edge-computing alternative. Nevertheless, Edge AI poses storage, computational, and efficiency challenges that must be addressed to support the deployment of compute-intense algorithms in embedded devices. To continuously increase the quality of their outputs, AI models are evolving into more complex algorithms, with extremely high memory and computing requirements that strain the resource capacity of edge low-power nodes. Aware of this challenge, the research community is studying the problem from different perspectives, mainly focusing on algorithmic optimizations or hardware accelerators. On one hand, the optimization of AI algorithms can reduce their memory need and computing complexity. On the other hand, the implementation of domain-specific hardware accelerators enables efficient executions of AI workloads by providing specialized resources designed to accelerate common kernels in AI inferences (e.g., matrix-vector multiplications).

Although optimization approaches tackling this problem from either an algorithmic or a hardware perspective exist, *hardware-software co-design methodologies* are key. Indeed, by employing a co-design strategy, hardware-aware

algorithmic transformations can be effectively harnessed in workload-aware hardware resources to retrieve real energy efficiency gains. In this context, I endorse the implementation of accuracy-driven co-design methodologies, as they can guarantee that the optimized design abides by user-defined output quality levels. Such a co-design optimization vision is the research focus of this thesis. First, I introduce the E^2CNNs methodology, an algorithmic-level transformation that builds ensembles of Convolutional Neural Networks (CNNs) to improve accuracy and robustness without increasing the initial memory and computing requirements. Then, I apply this methodology to different co-design strategies including codebook-based representations, approximate computing, and in-memory computing accelerators. The achieved results show that synergic combinations of hardware-aware application-level optimizations allow significant efficiency improvements in the evaluated AI benchmarks.

Keywords: Artificial intelligence, machine learning, deep learning, convolutional neural networks, embedded systems, internet-of-things, edge AI, energy efficiency, co-design, heterogeneous optimization.

Résumé

L'INTELLIGENCE artificielle (IA) est en train de révolutionner une vaste gamme d'applications industrielles et scientifiques en raison de ses nombreux avantages, notamment ses capacités d'auto-apprentissage, l'extraction de caractéristiques cachés à partir de données d'entrée, et sa flexibilité. Alors que le paradigme de l'informatique en nuage a été une référence pour les inférences d'IA au cours des dernières années, les progrès technologiques récents et les méthodes d'optimisation de l'IA préconisent et soutiennent un changement vers une alternative d'informatique en périphérie. Néanmoins, l'IA périphérique pose des problèmes de stockage de données, de calcul, et d'efficacité qui doivent être résolus pour soutenir le déploiement d'algorithmes à forte intensité de calcul dans les appareils embarqués. Pour améliorer en permanence la qualité de leurs résultats, les modèles d'IA évoluent vers des algorithmes plus complexes, avec des exigences extrêmement élevées en matière de mémoire et de calculs qui mettent à rude épreuve les ressources des nœuds périphériques à faible consommation d'énergie. Consciente de ce défi, la communauté des chercheurs étudie le problème sous différents angles, en se concentrant principalement sur les optimisations algorithmiques ou les accélérateurs matériels. D'une part, l'optimisation des algorithmes d'IA peut réduire leur besoin en espace mémoire ainsi que la complexité de calcul. D'autre part, la mise en œuvre d'accélérateurs matériels spécifiques à un domaine permet des exécutions efficaces des charges de travail de l'IA en fournissant des ressources spécialisées conçues pour

accélérer les noyaux communs dans les inférences de l'IA (par exemple, les multiplications matrice-vecteur).

Bien qu'il existe des approches d'optimisation abordant ce problème d'un point de vue algorithmique ou matériel, les méthodologies de *co-conception matériel-logiciel* sont essentielles. En effet, en employant une stratégie de co-conception, les transformations algorithmiques conscientes des enjeux matériels peuvent s'adapter pleinement aux ressources matérielles sensibles à la charge de travail afin de récupérer de réels gains d'efficacité énergétique. Dans ce contexte, j'approuve la mise en œuvre de méthodologies de co-conception axées sur la précision, car elles peuvent garantir que la conception optimisée respecte les niveaux de qualité de sortie définis par l'utilisateur. Cette vision de l'optimisation de la co-conception est l'objet de recherche de cette thèse. Tout d'abord, je présente la méthodologie E^2CNNs, une transformation au niveau algorithmique qui construit des ensembles de réseau de neurones convolutifs pour améliorer la précision et la robustesse sans augmenter les exigences initiales en matière de mémoire et de calcul. Ensuite, j'applique cette méthodologie à différentes stratégies de co-conception, y compris les représentations basées sur le codebook, le calcul approximatif, et les accélérateurs de calcul en mémoire. Les résultats obtenus montrent que les combinaisons synergiques d'optimisations au niveau de l'application en fonction du matériel permettent des améliorations significatives de l'efficacité dans les repères d'IA évalués.

Mots-clés : Intelligence artificielle, apprentissage automatique, apprentissage profond, convolutional neural networks, systèmes embarqués, internet des objets, edge AI, efficacité énergétique, co-conception, optimisation hétérogène.

Contents

Introduction

1.1 Artificial Intelligence

Artificial intelligence (AI) has been a research topic investigated by the computer science community for more than 60 years now. The term was first used in 1956 at the Dartmouth Conference, held at Dartmouth College, in the USA. Indeed, that conference is considered a milestone in the development of AI, although the roots of this approach can be traced even further. For example, the early foundations of AI can be found in the work of Alan Turing, who introduced the concept of *universal machine* [1] in 1936, paving the groundwork for the theoretical possibility of intelligent machines. Other key pioneers in the field of AI include Warren McCulloch and Walter Pitts, who introduced the concept of neural networks in 1943 [2]. In the years following the Dartmouth Conference, AI research gained momentum, and various algorithms and methods were developed. These include symbolic reasoning, expert systems, machine learning, and natural language processing. Since then, AI has received continuously growing attention, which led to significant progress and breakthroughs. An emblematic example of the incredible potential of AI in those years was represented by Deep Blue [3], an AI-based check player developed by IBM and able to defeat the world chess champion Garry Kasparov in 1997.

Over the years, numerous AI algorithms have been proposed. The *Logic Theorist* [4], developed by Allen Newell and Herbert A. Simon in 1956, was

one of the earliest AI programs, designed to prove mathematical theorems using symbolic logic and heuristic search. Two years later, Frank Rosenblatt presented the perceptron [5], a type of neural network algorithm aimed at mimicking the functioning of a biological neuron and able to recognize and classify patterns. The list of artificial intelligence algorithms grew faster and faster since then, with the introduction of more sophisticated and diverse AI algorithms encompassing areas such as machine learning, and, more recently, deep learning.

The high interest in AI is due to its capability to address a wide range of problems and challenges across a large pool of domains. A few examples of tasks and applications AI can efficiently handle include pattern recognition [6], predictive analytics [7], personalized recommendations [8], and fraud detection [9]. This ability makes these algorithms particularly appealing to solve tasks such as image recognition [10], speech recognition [11], and natural language processing [12]. In this context, AI finds application in fields like computer vision, voice assistants, and automated language translation, where Convolutional Neural Networks (CNNs) [13] represent widely investigated models. A more recent application of AI consists in providing personalized recommendations in various domains, such as e-commerce, streaming services, and content platforms, with AI-powered systems able to analyze user behavior and preferences. In all the presented applications, AI algorithms can also help in detecting anomalies in input data, identifying patterns of fraudulent behavior, and thus enhancing cybersecurity measures [14].

As a consequence of the vast range of opportunities offered by AI algorithms, several industries and scientific research areas are investing in this research topic. In healthcare, AI is currently used in medical imaging analysis, disease diagnosis, drug discovery, and personalized health monitoring using wearable systems [15–17]. Other sectors highly interested in employing AI as part of their software infrastructure include finance, manufacturing, education, and agriculture, as well as astronomy, genomics, drug discovery, climate modeling, and particle physics in scientific domains [18–20].

The reason why AI is so widely spread in almost every industrial and scientific application is due to the several advantages it provides when compared

to non-AI alternatives. First, AI can effectively handle complex and large-scale input data. In particular, deep learning models efficiently deal with high-dimensional and unstructured data (e.g., images, audio, or text), being able to extract relevant features to ultimately produce accurate predictions. Second, the learning ability of AI algorithms allows them to improve their accuracy over time. An embodiment of this approach, known as reinforcement learning [21], enables these models to handle evolving situations, making their predictions and decisions better and better over a certain number of simulations. Finally, AI algorithms scale well to different degrees of task complexity and can discover intricate patterns that traditional approaches may not easily discern. Nonetheless, despite the aforementioned advantages, it must be noticed that AI algorithms are not *always* superior to non-AI alternatives. Usually, the choice between AI and non-AI approaches may depend on several factors, such as the specific problem, the type and the amount of available data, the available hardware resources, and the need for results interpretability [22].

1.2 Convolutional Neural Networks

Among the plethora of AI, machine learning, and deep learning models proposed in the past years, this thesis focuses on Convolutional Neural Networks (CNNs) as target benchmarks, presenting different HW-SW co-design methodologies to optimize their execution from a resource, performance, energy, and accuracy perspectives. As detailed in the next paragraph, CNNs show different degrees of complexity, which make them good candidates to demonstrate the effectiveness of the proposed methodologies on a wide range of AI applications.

CNNs are indeed deep learning models that find applications in multiple fields, from computer vision [23,24] to personalized healthcare [25,26]. CNNs exhibit a layer-based structure, comprising convolutional, fully-connected, and pooling layers among the most common ones. These are combined in linear or more complex structures and enable the automatic extraction of features from input data (usually having spatial relationships, such as images), eventually producing abstract interpretations as output (e.g., recognizing objects, or classifying input samples). The parameters of CNN models, mainly

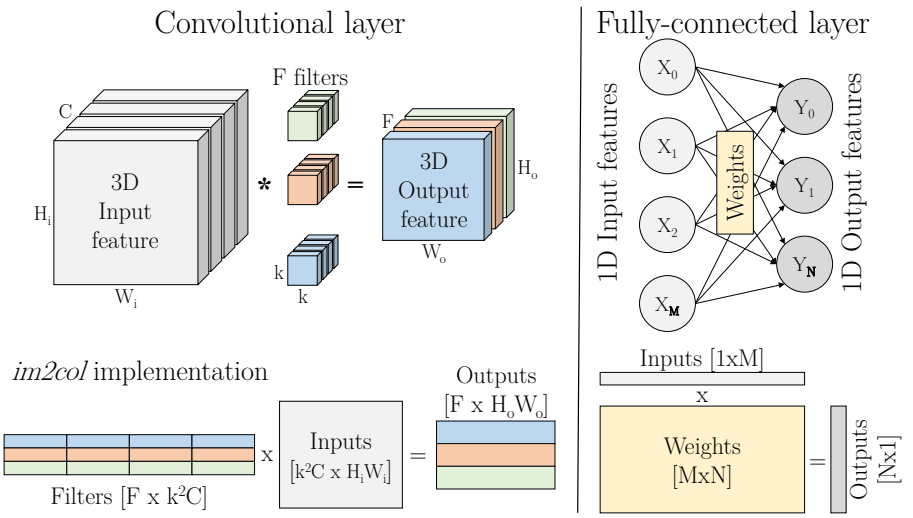


Figure 1.1: Computation of convolutional (left) and fully-connected (right) layers of neural network models. While fully-connected layers are naturally represented as matrix-vector operations, convolutions can be implemented in a similar way using an algorithmic transformation based on the im2col method (bottom).

used to perform convolutional and fully-connected layers, are referred to as weights. Conversely, the input and output features of each layer are known as activations.

Convolutional and fully-connected layers are the most compute-intense layers of CNNs, requiring the execution of millions of multiply-accumulate (MAC) instructions in recent models [27]. The former group together sets of learnable weights into multiple convolutional filters, which are then convolved over a region of the input data. Filters slide over the input features, producing a scalar output activation for each position covered. As a consequence, each filter produces as output a two-dimensional plane, and by employing multiple filters, convolutional layers can produce three-dimensional outputs. An example is illustrated in Figure 1.1(top-left), where three four-channel convolutional filters are applied to the four-channel input feature map to generate a three-channel output. Instead, in fully-connected layers, input and output elements, usually referred to as neurons, are connected

in a full-mesh topology by a matrix of weights, with output elements being computed using matrix-vector operations (Figure 1.1(top-right)).

Recent algorithms implement both convolutional and fully-connected layers as a series of matrix-vector multiplications. On one side, fully-connected layers are naturally represented as matrix-vector operations, with $w_{i,j}$ being the element of the weight matrix connecting the i-th input to the j-th output, as shown in Figure 1.1(bottom-right). On the other side, convolutions can be transformed to matrix-vector multiplications as well by properly reshaping input weights and activations. This offers computing advantages, especially when implementing them in HW accelerators, and it can be obtained using the *im2col* algorithm [28]. This approach indexes weights of entire convolutional layers as a matrix, including in each row the unrolled values of a filter. A schematic overview of this approach is depicted in Figure 1.1(bottom-left).

To perform their task (e.g., classification, detection, or segmentation), CNNs must undergo a process, called training, that allows them to properly tune their weights to achieve good performance. Training can be implemented in different ways, but the most common is supervised training [29]. First, a dataset comprising input samples enriched with corresponding labels (e.g., the class the sample belongs to, in the case of classification problems) must be collected. Usually, samples are pre-processed, by resizing them according to specific input constraints of the target CNN model, and by normalizing their values (e.g., fitting them to a specific data range). The dataset is then divided into training, validation, and testing sets. The first one is used to actually train the model, with the validation set being used to evaluate its performance during the process. The testing set is instead used only after the training is completed, to measure accuracy on new data, unseen during the training stage.

The weights of the CNN are first initialized, usually randomly and following Gaussian distributions. Then, the CNN model is fed with the samples in the training set and its output predictions are compared with the correct labels (forward pass). This comparison is used to measure, using a loss function, the discrepancy between the predicted outputs and the true labels. Gradients with respect to the computed loss are then evaluated using a process called

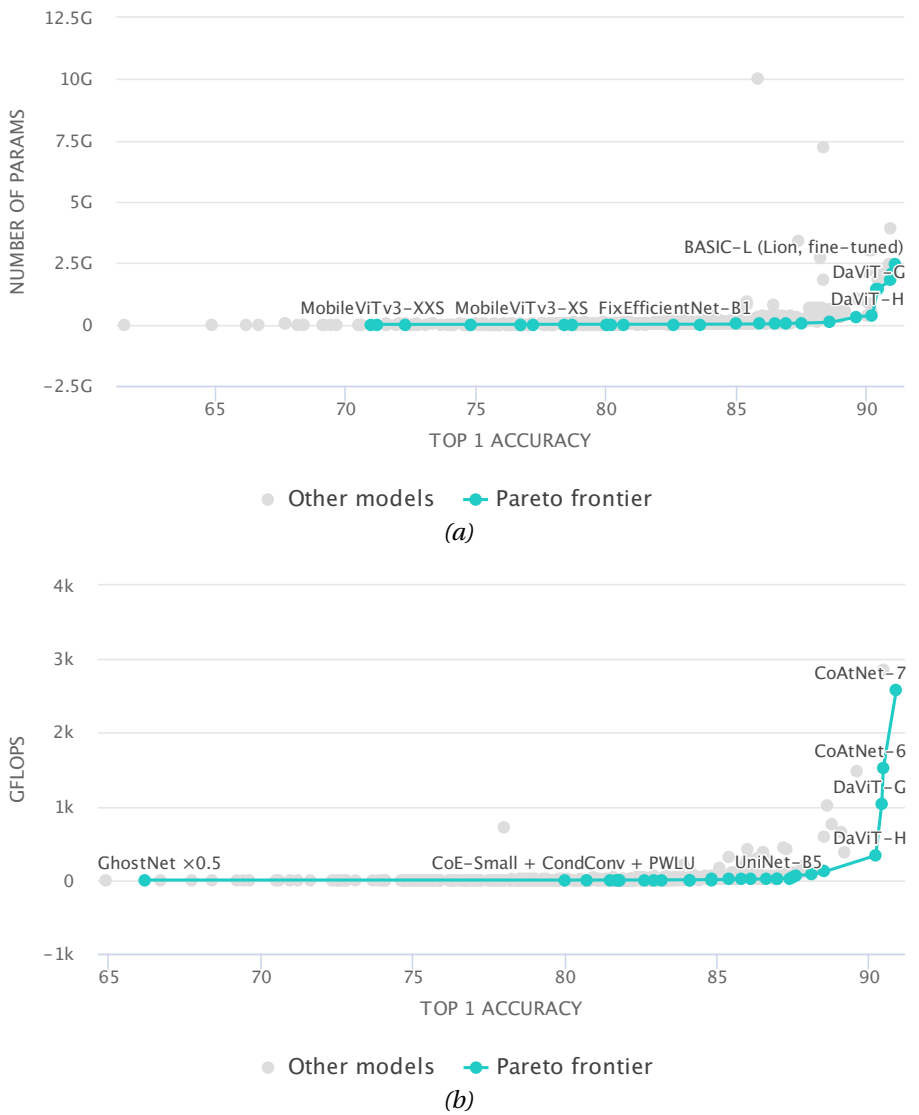


Figure 1.2: Number of parameters (a) and of floating-point operations (GFLOPS) (b) in state-of-the-art models designed for ImageNet classification. The highest accuracies are obtained with large and compute-intense models. Plots are extracted from *https://paperswithcode.com*

backpropagation, which propagates the gradients from the last layers of the CNN model to the first ones. Gradients are then used to update the weight values in each layer, in order to minimize the loss. Different algorithms have been proposed to implement the weights update, including Stochastic Gradient Descent (SGD), Adam, and RMSprop, among the most common. The validation set is then used to evaluate the new accuracy of the model, after the update of its parameters. The forward pass, the backpropagation, and the validation accuracy evaluation are repeated multiple times in an iterative procedure until the model accuracy converges or achieves acceptable performance. Once training is complete, the obtained model is tested on the so far unused testing set, to evaluate its performance on unseen data.

Figure 1.2 illustrates a summary of AI models proposed in the literature in the past years to solve image classification tasks. It compares them in terms of accuracy, as well as from memory and computing requirements perspectives. The best-performing architectures rapidly evolved to highly accurate implementations with extremely high memory and computing requirements. Indeed, the plots show that to achieve high accuracy, models tend to be larger and, in particular, increase their computing requirements, with billions of floating-point operations required in most models. This poses some challenges for the deployment of these models, especially when the execution is shifted from the cloud to the edge, as discussed in the following section.

1.3 Edge AI

In the era of rapidly evolving technologies and increasing data generation, edge computing has emerged as an alternative to cloud computing as a promising paradigm for distributed data processing [30]. On one side, both the cloud and the edge computing approaches aim to provide computing capabilities and solutions to the end nodes deployed for different applications in the field of the Internet of Things (IoT). On the other side, they differ in terms of architectural design, processing capabilities, latency, scalability, degree of privacy, and resource utilization. An overview of the two computing paradigms is illustrated in Figure 1.3.

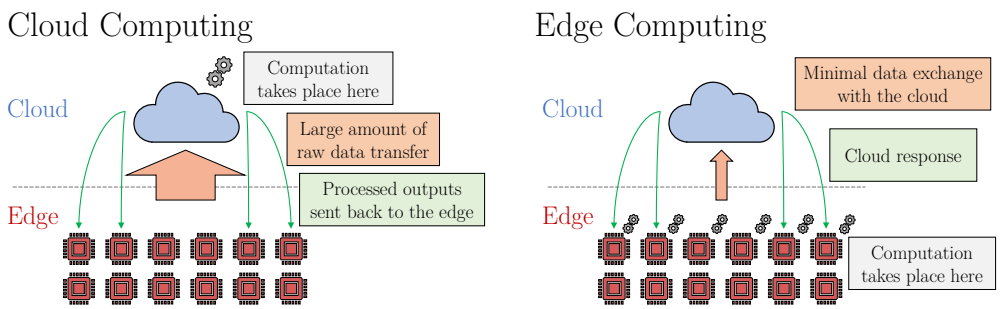


Figure 1.3: Comparison between the cloud computing and the edge computing approaches. By advocating data processing in the edge nodes, the amount of data collected at the edge and transmitted to the cloud is significantly reduced.

Cloud computing is a traditional approach that sees the end nodes mainly (or only) as data-collecting systems, leaving data processing to take place in large-scale data centers with high computing power and storage capacity (i.e., the cloud). Due to its centralized and high-performance architecture, cloud computing has the main advantage of providing high scalability and resource availability, being able to handle massive workloads and accommodating spikes in computing demand by provisioning additional resources [31].

Conversely, the edge computing paradigm is a decentralized approach that processes data at the network edge, where it is collected by the end nodes (or very close to it). Executing data processing workloads locally in the edge nodes offers several advantages. First, edge computing excels in low-latency and real-time scenarios because it minimizes the delay caused by transmitting data to remote servers. This makes it suitable for applications such as autonomous vehicles, industrial control systems, and, more in general, any real-time application. Second, for a similar reason, edge computing optimizes network bandwidth, since only the computed outputs, in contrast to the whole amount of collected inputs, are transmitted to the central cloud. Finally, it also addresses data security and privacy concerns by keeping sensitive data within the local network or device. The described advantages, as well as hardware technology and optimization methods advancements, are the core reasons for the rapid development of new edge computing solutions

in a wide range of applications. This also applies to artificial intelligence and goes under the name of *Edge AI*.

Nevertheless, new challenges arise in this context: one of the main limitations of edge computing resides in the hardware constraints of embedded devices, whose limited memory and computing resources can prevent the execution of large AI workloads. This problem is currently being investigated in the research community, which tackles it from different perspectives. It is also the main focus of this thesis, where I present different co-design methodologies combining application-level transformations with ad-hoc hardware optimizations to reduce memory, computing, and energy requirements in Edge AI inference.

1.4 Optimizations for Edge AI

Optimizations supporting Edge AI can be broadly divided into three main categories. First, continuous technology improvements and, in particular, CMOS technology scaling, allow new generations of embedded devices to be equipped with higher-capacity memories and faster computing units. However, these advancements are currently slowing down, while the requirements of AI workloads are increasing at a fast pace [32]. Therefore, Edge AI optimizations tackle the computing and efficiency challenges of edge computing from two other perspectives, widely investigated in the research community. This thesis shows how these two optimization paths can be merged to retrieve larger efficiency gains.

1.4.1 Application-level optimizations

On one hand, algorithmic-level optimizations aim at reducing the complexity of AI workloads. Several techniques have been proposed so far. *Quantization* is a popular method that reduces the precision of input and output operands from the traditional 32-bit floating-point format to more compact integer representations [33,34]. Common quantization levels include 8-bit and 16-bit schemes. Thus, quantization effectively reduces memory requirements, but can also improve efficiency as integer arithmetic requires simpler circuits than the ones required to manage floating-point formats. Targeting the same

objectives, *pruning* is another approach to reduce complexity, especially in machine learning and deep learning models [35, 36]. It removes specific computing elements from the original model and it is often applied to CNN models. Such an approach can be effectively applied due to the intrinsic redundancy of AI models, so that pruned architectures can still achieve acceptable accuracy levels. In the case of CNNs, pruning is usually applied to convolutional layers, either removing specific weights (fine-grain pruning) or removing entire convolutional filters (coarse-grain pruning). In addition to quantization and pruning methods, *weights encoding* [37, 38] and *weights clustering* [39] are other examples of strategies sometimes used to further shrink the memory needs of AI workloads. Most of these methods are orthogonal or complementary to each other and are hence often applied in synergy.

1.4.2 Hardware-level optimizations

On the other hand, hardware-level optimizations play a crucial role in achieving high-performance and energy-efficient Edge AI solutions and focus on providing AI applications with specific physical resources to efficiently execute the typical computing patterns of these workloads. To this end, ultra-low power processors have been designed for embedded systems to limit energy consumption. An example is the PULP platform [40], specifically designed to address the requirements of energy-efficient and high-performance computing in resource-constrained environments. It consists of a family of open-source processor cores optimized for ultra-low power consumption, that enable the implementation of parallel processing systems. The cores within the PULP platform are based on the RISC-V instruction set architecture [41], which enables customization and optimization. Targeting AI workloads, PULP also includes features and extensions that support AI algorithms, such as specialized instructions and hardware accelerators for efficient matrix operations, which are fundamental to many AI computations such as neural network inference.

In addition to low-power processors, dedicated hardware accelerators are largely employed in the field of Edge AI. Typical specialized units include Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs),

or Application-Specific Integrated Circuits (ASICs), which trade off computing flexibility for efficiency. Focusing on deep learning accelerators, two very popular classes of dedicated processing elements include systolic arrays [42] and neuromorphic accelerators [43]. The former consist of a grid of processing elements (PEs) interconnected in regular patterns to synchronously share data. These grids are often programmable, enable high levels of parallelism and data reuse, and, thanks to their structure, are particularly well-suited for energy-efficient executions of matrix-vector operations, which are indeed the core computing patterns of the majority of deep learning algorithms. Neuromorphic accelerators are instead inspired by the functionality of the human brain and excel in the execution of neural networks. By leveraging the inherent sparsity and irregularity of neural networks, and by simulating the sparse nature of neural activity, these accelerators reduce computational requirements and memory bandwidth, leading to significant efficiency gains. Sparsity-driven techniques, such as spike-based coding and event-driven processing, enable the selective and efficient processing of relevant information. Additionally, they also support on-chip learning and adaptability, allowing them to continuously learn and evolve with the data they process. A recent and revolutionary class of accelerators for Edge AI is represented by In-Memory Computing (IMC) devices [44, 45]. The IMC paradigm overcomes the traditional Von Neumann architecture by moving computation where data resides (or very close to it). By performing arithmetic and logic operations inside (or at the proximity of) the storage elements, IMC minimizes data movements and latency, enabling efficient and high-performance Edge AI solutions.

Another venue to reduce the energy cost of Edge AI workloads is the approximate computing paradigm [46–48]. The key idea is that inexact, yet simpler, arithmetic circuits can produce approximate outputs that can still lead to acceptable output quality levels. Thus, they trade off precision for faster and less energy-expensive executions. Common approximate operators include adders and multipliers, as highly stressed computing units in AI inferences [49, 50].

Finally, dynamic voltage and frequency scaling (DVFS) is a method that optimizes power consumption by adjusting the supply voltage and the oper-

ating frequency at which the system operates. Although not deeply investigated in the literature on AI optimization methodologies, few previous works have studied the impact of this technique from an energy-saving perspective [51, 52]. Since the voltage level is quadratically proportional to power consumption, its reduction can lead to significant energy savings, but it should be carefully tailored to avoid impacting performance and Quality of Service (QoS). In fact, errors can appear in the memory sub-system as a result of DVFS techniques: when reducing the voltage in SRAMs, stuck-at faults emerge as weaker bit-cells cannot be correctly written. Alternatively, when reducing the operating frequency in DRAMs, the resulting lower refresh rates can make bit-cells lose their content.

1.5 Trading-off accuracy for efficiency

Most of the optimization methods presented in the previous sections introduce data approximations or computation errors. For example, quantization and clustering adjust the weights of AI models, forcing them to assume specific values, hence being equivalent to a form of data approximation. Similarly, inexact operators introduce approximations in the performed computation, while pruning approximates the input-output relation by reducing the complexity of the involved functions. Aggressive voltage scaling can introduce stuck-at faults when applied to memory elements, as weaker bit-cells do not receive enough energy to flip their content during write operations.

As a result, different degrees of inexactness usually affect inferences in Edge AI when the presented optimization methods are applied. Nevertheless, the research community has demonstrated that, up to a certain degree, these techniques do not significantly affect output quality [53–55]. In fact, the redundant structure and sparse nature of most ML models (e.g., random forests and neural networks) make them intrinsically tolerant toward a certain level of inexactness. This is why it is possible to retrieve essential efficiency improvements by introducing specific magnitudes of approximation so that the desired QoS can still be achieved. Importantly, the algorithmic characteristics of ML models cannot provide a designer with a priori knowledge of the accuracy impact of a specific error density or noise level. In other words, it is not possible to determine, at design time, the accuracy degradation that

a certain model experiences when one of the aforementioned optimization approaches is employed. For example, voltage reductions may cause memory errors, which randomly affect memory bit-cells. As a consequence, the impact of these faults will be different based on the affected cells: for example, errors affecting the most significant bits (MSb) of a word usually introduce larger variations than errors affecting the least significant bits (LSb).

The methodologies for optimizing Edge AI applications that I present in this thesis take into account and address this challenge: I will show different strategies to optimize AI workloads, combining algorithmic-level and hardware-level methods to increase achievable gains. In all proposed design methodologies, I will include accuracy constraints to drive the optimization procedures. This approach, sometimes overlooked in state-of-the-art studies, enables the optimizations of AI applications from memory, performance, and energy perspectives, while also abiding by user-defined accuracy levels. As previously discussed, I consider CNN models as target benchmarks for evaluating the proposed Edge AI methodologies. Nonetheless, the majority of the presented solutions and techniques can be naturally extended to different models in the field of AI.

Effective integration between hardware and software is crucial for optimizing Edge AI systems from energy and performance points of view. On one hand, hardware-aware software optimization, such as algorithmic transformations and quantization can adjust AI models in order to maximize the utilization of hardware resources and effectively leverage them. On the other hand, software-aware accelerators can be designed to efficiently implement specific computing patterns of the target workload or application domain. Without such a co-design vision, the risk of not being able to fully exploit the potential of the applied optimization is high. For example, very fine-grained quantization levels cannot reach the desired shrink of memory requirements if the target platform only supports 32-bit data. Dually, reducing the voltage to save energy cannot be a practical solution if the application is not robust enough to cope with the potential memory errors deriving from this technique. Starting from these considerations, the next chapters describe co-design methodologies targeting domains where different classes of hardware resources are available.

First, I introduce the concept of *Embedded Ensembles of Convolutional Neural Networks (E^2CNNs)*, an algorithmic-level transformation I propose to improve robustness against errors in neural network models. E^2CNNs combines pruning and replication to construct ensemble-based models that benefit the higher accuracy and resiliency of state-of-the-art ensembles, but without increasing the requirements of the baseline single-instance model. Then, I take advantage of the characteristics of E^2CNNs to retrieve high-efficiency gains in a wide spectrum of AI co-designs.

I initially focus on general-purpose processing units as target hardware resources. In this case, ad-hoc hardware accelerators may not be available, thus posing a limit to the pool of algorithmic-level optimizations that can be effectively exploited in hardware. I propose a methodology based on codebook-based optimizations, a class of algorithmic-level transformations that effectively reduce the computing and memory requirements of AI models. By tightly limiting the number of unique weight values, they allow the storage of representative parameters in small look-up tables (i.e., codebooks) containing a limited number of floating-point entries. AI models are then represented as low-bitwidth indexes of such codebooks, enabling model compression while preserving floating-point inference. I introduce a novel methodology that employs an E^2CNNs design and finds highly beneficial codebook schemes, trading off accuracy for model compression in codebook-based models.

Then, I consider the availability of approximate hardware as a design choice to reduce energy in Edge AI. Indeed, previous studies have demonstrated that, up to a certain degree, AI models can tolerate noisy input data and inaccurate intermediate results, ultimately being still able to produce acceptable output qualities. Nonetheless, when dealing with arithmetic inexact operators, a judicious use of approximation is crucial to limit accuracy degradation. Determining the magnitude of inexactness that can be introduced is a challenging task because the implementation of inexact operators is often decided at design time when the application and its robustness profile are unknown. The result is a risk of over-constraining or over-provisioning the hardware. To bridge this gap, I propose a two-stage optimization that initially optimizes the target model, applying E^2CNNs in conjunction with a heterogeneous

quantization strategy. This phase reduces the bitwidth of input and output operands enhancing at the same time error resiliency, so that inexact operations can be performed as frequently as possible. Then, a sensitivity analysis is performed and used to drive the mapping of computing kernels into either exact or inexact hardware.

Finally, I target In-Memory Computing (IMC) accelerators, proposing different implementation and algorithmic optimization approaches to maximize energy savings when running AI inferences. In fact, IMC supports the efficient execution of data-centric workloads, such as those characterizing AI algorithms. These accelerators provide computing capabilities as part of the memory array structures, bringing the processing elements inside storage. By doing so, IMC minimizes the cost of data access and enables highly parallel computations 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. To tackle this challenge, I introduce a novel optimization heuristic, which tailors the quantization levels according to both memory design characteristics and workloads considerations. I also show how lightweight hardware support is required in the proximity of storage elements to increase computing parallelism and to ensure overflow-free arithmetic under strict bitwidth constraints. Finally, I combine most of the presented techniques and methodologies in a more complete co-design approach. On the one hand, I show that highly effective error detection and mitigation strategies can be implemented in IMC devices with little extra hardware. On the other hand, I use algorithmic transformations including E^2CNNs and quantization to increase the robustness of AI models against errors. In doing so, more aggressive voltage scaling techniques can be applied to run the accelerator relying on ultra-low supply voltage levels. While this approach inevitably introduces errors in the memory arrays, the higher degree of resiliency exposed by algorithmic optimization minimizes the impact of such errors on the output quality of the model.

2

Embedded Ensembles of Convolutional Neural Networks

Herein, I present *Embedded Ensembles of Convolutional Neural Networks* (*E^{2}CNNs*), a methodology to build memory- and compute-constrained ensemble-based models. First, in Section 2.1, I include a brief analysis of CNN and ensemble-based models, discussing the two alternative implementations in terms of timing and resource requirements and from an error resiliency perspective. Then, I detail the E^{2}CNNs methodology in Section 2.2, describing how to construct and train constrained ensembles, analyzing their benefits with respect to state-of-the-art ensembles of CNNs. Finally, in Section 2.3, I evaluate the E^{2}CNNs design by comparing different E^{2}CNNs configurations to the corresponding baseline single-instance CNNs. This methodology has been presented in [56].

2.1 Introduction

2.1.1 Robustness of CNN models

In many real-world scenarios, input data is often corrupted by noise or errors. Measurement imprecision, interference, memory faults, or incorrect data transmission are just a few examples of data corruption sources [57–59]. In such situations, it is crucial to have robust algorithms that can tolerate such inaccuracies and still provide accurate and reliable outputs. In this regard, machine learning models happen to be highly resilient to input noise and data approximations [48, 53]. In particular, CNN models well tolerate errors

Table 2.1: Yield analysis showing the percentage of simulated chips with accuracy degradation limited to 5 % for an industrial CNN model used for capsule recognition at different sub-nominal voltage levels. Error rates at different voltage supplies are derived from [51].

Voltage	850 mV	750 mV	700 mV	650 mV	600 mV
Error rate	0	1.3×10^{-5}	1.0×10^{-4}	7.0×10^{-4}	2.2×10^{-3}
Floating-point	100.0%	11.2%	0.0%	0.0%	0.0%
Fixed-point 8/16	100.0%	100.0%	99.5%	66.3%	0.2%

in the processed data, thus being able to offer acceptable output quality, even in non-ideal conditions [54]. Nevertheless, these models are nowadays employed in a vast range of applications, that typically require different degrees of robustness and must abide by specific Quality of Service (QoS) metrics. For example, computer vision, speech recognition, or natural language processing applications can tolerate a certain degree of QoS degradation without significantly affecting their overall performance and user experience. On the contrary, the quality of the output is a must in sensitive or safety-critical applications, such as medical diagnosis, autonomous driving vehicles, financial forecasting, and industrial automation.

To investigate the robustness of CNN models against errors, my colleagues and I have evaluated the impact of data representation in a proprietary CNN model used for coffee capsule recognition [60]. In particular, our analysis considered memory errors due to sub-nominal voltage supplies in the memory sub-system, producing stuck-at faults in the input data, also impacting the whole inference execution. We have conducted experiments varying the voltage level to evaluate the accuracy degradation at different error rates. For each of them, we have considered 1000 random error maps which emulate different fabricated memory chips. Finally, we have performed a yield analysis to measure the percentage of chips that were able to achieve accuracies within 5% of the baseline. A simplified, yet not reductive, overview of the results obtained in [60] is presented in Table 2.1, comparing the yield achieved by a floating-point model with the one obtained quantizing weights

and activations using 8 and 16 bits, respectively (i.e., a scheme I refer to as 8/16 quantization level). These results provide multiple insights: first, the yield obtained at a nominal voltage (i.e., 850 mV) confirms that, if properly designed, quantization has a negligible impact on accuracy. More precisely, accuracy evaluations showed that the accuracy degradation with respect to the floating-point baseline is negligible when employing an 8/16 quantization scheme. Second, results indicate that a fixed-point format is significantly more robust than floating-point representations, achieving a yield of 99.5% at 700 mV, where instead none of the chips employing a floating-point representation can reach the target accuracy level. The higher sensitivity toward memory errors of the floating-point implementation can be explained by considering the peculiar memory representation of floating-point values. In fact, the distinction between sign, exponent, and mantissa bits may be critical if errors affect certain bit positions: for example, errors affecting the sign bit produce dramatic deviations in high-magnitude floating-point numbers. In a similar way, errors affecting the exponent bits can transform relatively small values into high-magnitude ones, and vice-versa. In addition, specific floating point codes are used to represent $\pm\infty$ and not-a-number (*NaN*) values. As a consequence, if errors make them appear, these values propagate through the CNN layers, preventing arithmetic computation and thus producing unreliable outputs. In contrast, fixed-point formats do not suffer similar effects, as, in the worst case, errors affecting the sign or the most significant bits can have a significant, but not so critical, impact.

As a result, this analysis shows that fixed-point representations are key for the resiliency of CNN models and, if properly dimensioned, these formats do not even affect the baseline floating-point accuracy. In line with the results presented in related works [54, 55], this analysis demonstrates the intrinsic resiliency of (quantized) CNN models. Nevertheless, above a certain threshold, the error density results in high-magnitude data perturbations that dramatically degrade output quality. Error protection mechanisms and algorithmic strategies to improve the robustness against errors have a double target. On one hand, mitigating the impact of errors by limiting noise or introducing error detection mechanisms (e.g., parity check or memory error correction codes) is crucial in safety-critical applications. On the other hand,

an increased robustness against errors can be leveraged to enable more aggressive optimization strategies, ultimately achieving improved energy efficiency levels for a minimal accuracy cost, as discussed in Chapter 1.

While several approaches to improve the accuracy and robustness of CNN models exist [61–63], I am now going to focus on ensembling methods [64–66], proposing a novel way to construct ensemble-based models for Edge AI inference.

2.1.2 Ensembling

In machine learning, ensembling is a technique that combines together multiple algorithmic instances to create a more robust and accurate model [67]. The core idea is that by combining the predictions of several *weak* models, the errors and biases of individual models can be offset, leading to a more precise and reliable overall prediction.

As deeply detailed in [67], there are several ways to perform ensembling in machine learning applications, including:

- *Bagging*. It consists in training multiple models on random subsets of the training data and then combining their predictions to produce the output result. Bagging is often used with decision tree models, creating a random forest ensemble.

- *Boosting*. In this technique, multiple weak models are trained sequentially, with each new one trained on the residuals of the previous model. The final prediction is then a weighted average of the predictions of all the trained instances.

- *Stacking*. It involves training multiple instances and using their predictions as input features for a higher-level model, making the final prediction. This approach can be particularly effective when the individual instances are trained on different aspects of the data, allowing the higher-level model to learn from a wider range of features.

Ensembling is a powerful technique that has been used with great success in many areas of machine learning, including image recognition, natural language processing, and speech recognition [64, 65].

While bagging mainly operates on dataset splits and, like boosting, is commonly used to build random forests, stacking is instead a more general ensembling method that has been also applied to CNN models [66], and hence considered in this chapter. In this regard, it employs multiple CNN instances, possibly sharing different structures, and combines together their individual predictions to compute the ensemble output. Depending on the objective task, aggregation can be implemented either as a simple average of individual predictions of each CNN instance in the ensemble or using those predictions to feed (and tune) an additional meta-model. The meta-model, typically a very simple machine learning model, eventually uses these predictions as input features to produce the final output.

One of the key advantages of stacking is that it can handle complex relationships between features and target variables. By combining the predictions of multiple models, stacking enables the capture of a wider range of patterns and features from the input data, leading to improved accuracy and generalization performance. Even more, ensemble-based architectures also increase resiliency against errors. Figure 2.1 illustrates this concept by comparing the effect of an error in a single-instance CNN model (left) and in an ensemble of four CNNs (right). This simple example shows that an error affecting the first layer of the single-instance CNN model results in a wrong classification of the input image. On the other hand, the same error affecting the first instance of the ensemble still produces a misclassification in the first CNN instance. Nonetheless, the wrong prediction can be mitigated by the correct predictions of the other three instances, ultimately allowing the ensemble to correctly classify the input sample.

Stacking but, more in general, any ensembling method, is computationally expensive, requiring careful tuning of the base models and, if included, of the meta-model. Let's consider the example depicted in Figure 2.1, assuming the four models in the ensemble-based design share the same structure. The higher accuracy and resiliency of the ensemble-based architecture are

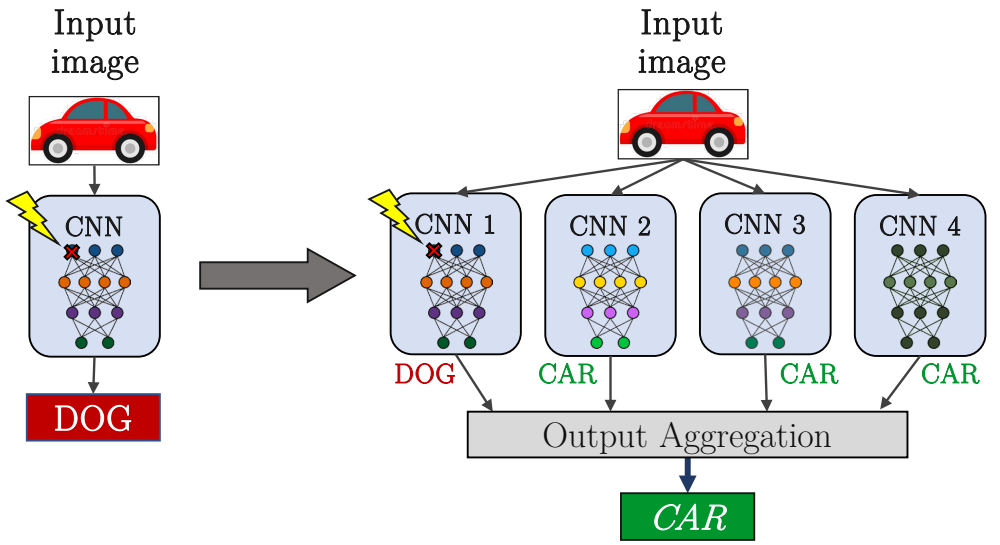


Figure 2.1: Single-instance CNN model (left) and a state-of-the-art ensemble-based alternative comprising four CNN instances (right). While leading to misclassification in the single-instance model, the same error can be mitigated in an ensemble-based implementation.

achieved at the cost of memory and computing overheads. In particular, memory requirements increase by 4× because four CNN models have to be stored in memory. Similarly, inference runtime also increases by a similar factor, as four CNN models must be evaluated. In general, overhead may be even larger, since in some cases tens or hundreds of CNN models are combined together [68, 69].

Therefore, while the accuracy and robustness gains of state-of-the-art ensembles are key for most applications, their actual deployment in edge devices is often prohibitive, due to the memory, timing, and energy constraints of embedded systems. To address this limitation, I propose Embedded Ensembles of Convolutional Neural Networks (E^2CNNs) [56], a methodology to build and deploy ensembles in edge AI applications. As the name suggests, it targets CNN models, although the key concept of this methodology could be applied to different machine learning models.

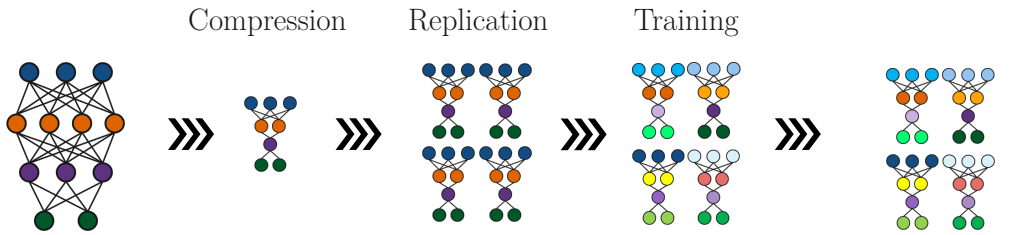


Figure 2.2: E^{2}CNNs methodology. The baseline CNN model is first compressed to reduce its complexity. Then, the obtained (pruned) structure is replicated multiple times to build an ensemble that has the same properties as the baseline model. Each instance is trained from initial random weight initialization, resulting in different trained weights distribution, key to improving accuracy and robustness.

2.2 E^{2}CNNs methodology

E^{2}CNNs transforms a target CNN model into an equivalent ensemble-based architecture. Targeting edge devices, the objective is to construct ensembles of CNN models that do not increase the memory and computing requirements of the initial single-instance network. To achieve this goal, E^{2}CNNs combines pruning and replication to construct ensembles that benefit from the higher accuracy and robustness of state-of-the-art ensembles, but without increasing baseline requirements.

2.2.1 Building the ensemble

The proposed methodology to build E^{2}CNNs is summarized in Figure 2.2. The first step is to compress the baseline single-instance structure to reduce its complexity. In particular, the input can be an *untrained* CNN architecture, and the compression stage reduces the number of parameters (i.e., weights and biases of convolutional and fully-connected layers), as well as the number of multiply-accumulate (MAC) operations required to execute the network. In general, both metrics are reduced by a factor N. Compression is performed via filter pruning [35]. Since the input single-instance structure is an untrained architecture, random filter pruning is used. This approach removes random filters from the convolutional layers of the input

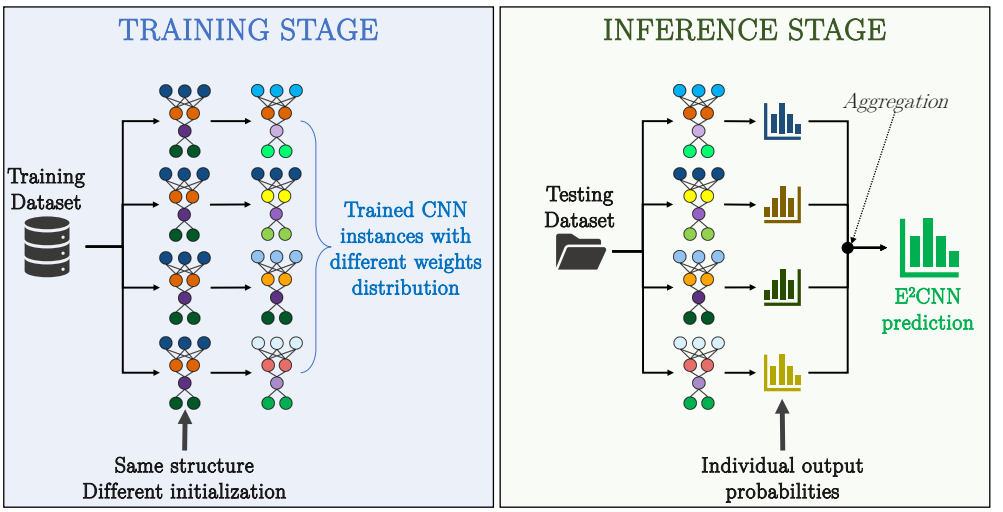


Figure 2.3: E^{2}CNNs training (left) and inference (right) stages. Training is performed offline, before deploying the ensemble in the target application. At inference time, the same sample is processed by all trained instances, with a final aggregation step producing the E^{2}CNNs prediction.

CNN. While more advanced filter pruning methods have been proposed [36], the pseudo-randomness selection of filters to remove makes this approach one of the few approaches suitable for untrained CNN structures. To avoid over-compressing certain layers and under-compressing others, filter pruning homogeneously compresses the whole architecture.

The resulting pruned structure exhibits $N\times$ fewer parameters and $N\times$ less MAC operations than the initial model. As a consequence, the obtained pruned CNN can be replicated N times to build an ensemble of CNNs with N instances, without incurring any memory or computing overheads with respect to the initial single-instance baseline.

Before their deployment in the field for inference executions, the N models must first be trained. Training is performed offline, before deploying E^{2}CNNs in the target device. A high-level scheme showing the two distinct phases is presented in Figure 2.3. Each pruned CNN is independently trained on the entire training dataset, starting from a random initialization of the model

Table 2.2: LeNet5 vs. LeNet5-based 2-E^{2}CNNs structure

LeNet5	In/Out Feature Size	MACs	Params
Convolutional	32×32×1 ⟶ 28×28×6	117,600	156
Max Pool	28×28×6 ⟶ 14×14×6	-	-
Convolutional	14×14×6 ⟶ 10×10×16	240,000	2,416
Max Pool	10×10×16 ⟶ 5×5×16	-	-
Convolutional	5×5×16 ⟶ 1×1×120	48,000	48,120
Fully-Connected	120 ⟶ 84	10,080	10,164
Fully-Connected	84 ⟶ 10	840	850
	Totals	**416,520**	**61,706**
2-E^{2}CNNs instance	In/Out channels	MACs	Params
Convolutional	32×32×1 ⟶ 28×28×4	78,400	104
Max Pool	28×28×4 ⟶ 14×14×4	-	-
Convolutional	14×14×4 ⟶ 10×10×10	100,000	1,010
Max Pool	10×10×10 ⟶ 5×5×10	-	-
Convolutional	5×5×10 ⟶ 1×1×80	20,000	20,000
Fully-Connected	80 ⟶ 84	6,720	6,804
Fully-Connected	84 ⟶ 10	840	850
	Totals	**205,960**	**28,848**

weights. In this way, the resulting trained CNN instances will show different weights distributions, which ultimately allow E^{2}CNNs to improve its generalization capabilities and robustness. Notice that an ensemble where all the CNN instances have the same weight values will perform exactly like each individual instance would perform alone. Finally, the trained models are deployed in the field. To act as an ensemble, the individual predictions of each CNN instance are combined together using a simple output average.

2.2.2 Example: LeNet5 vs. LeNet5-based E^{2}CNNs

This section presents an example showing a very simple CNN model being transformed into an E^{2}CNNs equivalent. For simplicity, this example considers LeNet5 [70] as the baseline single-instance CNN, and an E^{2}CNNs

implementation composed of two instances (in the rest of this thesis, I use the N-E^{2}CNNs notation to refer to ensembles of N instances).

The structure of LeNet5 is reported in Table 2.2(top). It consists of three convolutional layers and two fully-connected (FC) layers, with two additional pooling layers to reduce input dimensionality. The second column of the table reports the size of input and output features for each layer, in the form $W \times H \times C$, with W, H, and C being the width, height, and depth of the corresponding feature maps, respectively. Next, the last two columns show the number of MAC operations executed in each layer and the number of weights required for their execution. Pooling layers have no parameters and their complexity is negligible compared to convolutional and FC layers. In total, LeNet5 executes more than $400K$ MAC operations per inference and needs more than $60K$ parameters.

Building 2-E^{2}CNNs requires halving both the memory and the computing requirements of LeNet5. The result of the pruning stage is shown in Table 2.2 (bottom). Filter pruning reduces the number of filters in the three convolutional layers, from the baseline 6, 16, and 120 filters, to the 4, 10, and 80 filters applied by the pruned model. The reduction of the number of filters has a positive effect on both memory size (since fewer filters must be stored) and performance (since a lower number of filters must be applied to the input features). Reducing the number of convolutional filters in layer i positively impacts both layer i and layer $i+1$. In fact, fewer filters in layer i produce smaller output feature maps (i.e., outputs have a lower number of channels). As a consequence, this has an effect on layer $i+1$, because each filter can be smaller, as it must be applied to a reduced number of input channels.

In conclusion, the table shows that the pruned instance used to build 2-E^{2}CNNs has approximately halved the number of MAC operations *and* the number of weights. Therefore, when employing two instances, there will be no overhead when compared to the baseline LeNet5 implementation.

2.2.3 Selecting the E^{2}CNNs cardinality

This section concludes the description of the E^{2}CNNs methodology discussing how to determine the *cardinality* of the generated ensemble. The

term cardinality is used to indicate the number N of instances included in E^{2}CNNs. I will show in Section 2.3 that E^{2}CNNs of different cardinality achieve different accuracy levels and exhibit different robustness degrees, thus making the selection of the E^{2}CNNs cardinality N an important design choice.

In general, E^{2}CNNs implementations of different cardinality could be implemented and evaluated, eventually selecting the best-performing one. Nevertheless, training several CNN instances can be a time-consuming process and consumes a large amount of energy. Moreover, the design space can be extremely complex if N is not bounded or constrained. In [56], I also propose a heuristic approach that automatically selects the cardinality of the ensemble based on a target application and an expected error rate. An overview of the E^{2}CNNs selector is summarized in Algorithm 1. It receives as input the baseline single-instance CNN, a user-defined set of possible quantization levels to be applied, an accuracy threshold, and an expected error density. First, the most aggressive quantization scheme that still preserves accuracy to the user-defined level is employed. Experimental evaluations on multiple benchmarks revealed that the 8/16 quantization level (i.e., 8-bit weights and 16-bit activations) typically results in accuracy within 1% with respect to the floating-point baseline. Then, the single-instance model is iteratively pruned, reducing its complexity by 2× at each step. This approach constrains the cardinality N to be multiple of 2, thus evaluating E^{2}CNNs comprising 2, 4, 8, ..., instances. Although N can be in practice any integer value greater than 1, this heuristic aims to reduce this exploration's complexity. Experimental observations showed that only a highly-over-designed single-instance model can be transformed into an ensemble with a cardinality $N > 8$. Therefore, most commonly, only E^{2}CNNs implementations of 2, 4, or 8 instances are the ones considered by the heuristic. When a pruned CNN architecture is established, it is then trained, and the accuracy is evaluated. Finally, it is compared with a threshold, empirically determined based on the required robustness level needed. In the case of expected high error densities (or high-magnitude noise levels), the need for additional robustness becomes more crucial and the heuristic allows accuracy drops up to 5% in the pruned CNN instance. Conversely, in the case of low expected error rates, the accuracy degrada-

tion is limited to just 1%. The reason is based on empirical evidence and motivated by the fact that ensembles with high cardinality trade off a lower accuracy (since each instance is pruned more) for a higher robustness against errors. In other words, increasing cardinality initially improves accuracy. Yet, at a certain point where the pruning required to build the ensemble becomes too aggressive, the accuracy of the ensemble starts dropping. Conversely, robustness always tends to increase with cardinality. For this reason, relaxing the accuracy threshold in the case of high noise levels supports the construction of higher cardinality (and hence more robust) E^{2}CNNs architectures. Consequently, more conservative thresholds limit the possibility of having high-cardinality ensembles in favor of E^{2}CNNs designs with higher accuracy in the absence of significant noise levels.

2.3 E^{2}CNNs achievements

2.3.1 Experimental set-up

The E^{2}CNNs methodology is evaluated on a pool of benchmarks of different complexity. The following CNN models and datasets are included in the study:

- LeNet5 [70] on the MNIST dataset [71]
- AlexNet [72] on the CIFAR-10 dataset [73]
- GoogLeNet [74] on the EuroSAT dataset [75]
- MobileNet [76] on the EuroSAT dataset [75]
- A proprietary CNN (*CapsuleNN*) used for coffee capsule recognition

CapsuleNN is a proprietary network with a structure similar to AlexNet, but specifically designed to classify coffee capsules. This is the same model my colleagues and I have analyzed in [60], evaluating its resiliency against errors. LeNet5 is a tiny CNN model, introduced in the literature several years ago. AlexNet has significantly larger convolutional layers, resulting in a much more compute- and memory-intense network to execute. GoogLeNet and

Algorithm 1 E^{2}CNNs heuristic. In the first stage, a certain quantization level preserving the baseline floating-point accuracy is selected. Then, an iterative heuristic search evaluates the accuracy of different pruned CNNs. The resulting E^{2}CNNs cardinality is determined based on the accuracy and considering an expected error density.

```
1: procedure BUILDER(ErrorRate, QuantLevels, AccMin)
2:     i ← 0
3:     repeat
4:         Quant ← QuantLevels[i]
5:         NewModel ← Quantize(Model, Quant)
6:         if NewModel.Acc ≥ AccMin then
7:             Model ← NewModel
8:         end if
9:         i++
10:    until NewModel.Acc < AccMin
11:    Accuracy ← Model.Acc
12:    if ErrorRate >= 0.0001 then
13:        DropThreshold ← 5%
14:    else
15:        DropThreshold ← 1%
16:    end if
17:    repeat
18:        NewModel = Model.prune(compression = 2x)
19:        NewModel.Train()
20:        drop ← Accuracy − NewModel.Acc
21:        if drop < DropThreshold then
22:            Model ← NewModel
23:        end if
24:        i++
25:    until Drop ≥ DropThreshold
26: end procedure
```

MobileNet are instead much deeper networks (i.e., they both contain more than 20 convolutional layers, while LeNet5 and AlexNet only contain three and five, respectively) which also include normalization layers. Additionally, GoogLeNet uses inception blocks, while MobileNet uses separable convolutional layers.

The models are trained and quantized in PyTorch. Quantization is implemented using an approach known as *fake quantization* [77]. It forces weights and input/output activations to assume values representable in the target quantization scheme while maintaining a floating-point notation. As a consequence, built-in functions like convolutional or fully-connected layers use floating-point precision for their execution, although inputs and outputs can be updated between consecutive layers to simulate quantization. Therefore, to simulate real fixed-point arithmetic, a CNN model trained and optimized in PyTorch is then stored into a binary file, so that its weights can be easily imported into a C++ inference solver that I have developed to retrieve more accurate accuracy evaluations. The inference solver can execute convolutional, fully-connected, pooling, and normalization layers, to effectively implement different CNN architectures. CNN models are defined by connecting these layers properly, thus implementing their structure to simulate their behavior. 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 specific header file that represents the CNN model. This file also defines the size of the input buffer and two intermediate ping-pong buffers storing input and output activations of each layer. Even more, each layer's weight and bias values is used to generate pointers to the array of weights retrieved from the input binary file. In this way, each layer defined in the structural implementation of the CNN model can access the correct set of parameters for its execution. In a similar way, input binary images are imported into the C++ framework to execute inference simulations.

In addition to error-free accuracy estimations, the C++ solver also implements an error model simulating SRAMs working at sub-nominal voltage levels and eDRAMs with reduced refresh rates. Both operating conditions enable energy savings in the memory, but introduce stuck-at faults (in SRAMs) or bit-flips (in eDRAMs). Table 2.3 reports the error rates and the read/write energy cost

Table 2.3: Energy consumption per access (pJ/access) and bit error rate for an SRAM built on a 40 nm CMOS process at different voltage levels. Error rates at different voltage supplies are derived from [51].

	Read	Write	Error Rate
850 mV	9.447	5.868	-
750 mV	7.572	4.703	1×10^{-5}
700 mV	6.766	4.202	1×10^{-4}
650 mV	6.047	3.756	7×10^{-4}
600 mV	5.416	3.364	2×10^{-3}

corresponding to sub-nominal supply voltage levels in SRAMs as illustrated in [51]. When simulating errors affecting eDRAMs, the refresh rate is reduced to reproduce error densities comparable to the ones used to evaluate SRAMs.

The use of these error densities allows me to evaluate the robustness improvements of E^{2}CNNs while, at the same time, estimating potential energy savings by considering the energy of read and write accesses in memory elements corresponding to the considered sub-nominal voltage levels.

2.3.2 Accuracy improvements

Considering the benchmarks presented in Section 2.3.1, I herein present an analysis of the accuracy achieved in E^{2}CNNs and in the corresponding single-instance baselines, for error-free simulations. The highest accuracy achieved by each benchmark among the different evaluated architectural designs (i.e., single-instance CNNs or E^{2}CNNs alternatives) is highlighted in bold in Table 2.4. Results indicate that even if E^{2}CNNs has been initially designed to improve error resiliency, it also enables accuracy improvements when compared to single-instance CNNs. The obtained accuracy gain is not due to a higher number of parameters like it generally happens in state-of-the-art ensembles. In fact, Section 2.2.1 showed that the number of parameters in any E^{2}CNNs design is comparable with the one of the input single-instance CNN. The higher accuracy is just the result of the improved generalization capabilities offered by an ensemble-based architecture.

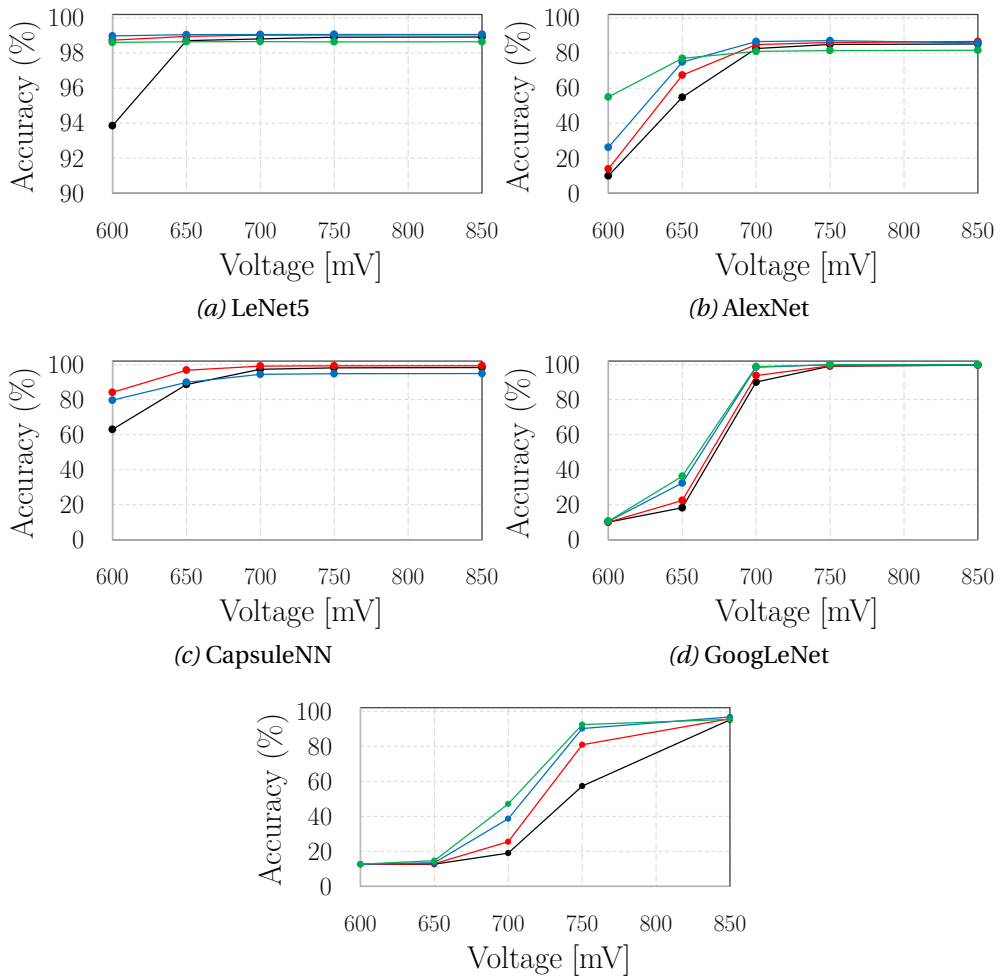
Table 2.4: Accuracy comparison between the original (quantized) CNNs and three different E^{2}CNNs embodiments (error-free evaluations)

Model	Original (%)	2-E^{2}CNNs (%)	4-E^{2}CNNs (%)	8-E^{2}CNNs (%)
LeNet5	98.91	99.05	**99.06**	98,64
CapsuleNN	98.29	**99.34**	94.87	-
AlexNet	85.20	**86.60**	85.80	81.60
GoogLeNet	99.70	99.70	**100.00**	99.90
MobileNet	95.125	95.75	**96.75**	95.37

The obtained results also suggest that, in error-free inference simulations, increasing the cardinality N of E^{2}CNNs will eventually result in accuracy degradations for high cardinality values. This effect manifests clearly in LeNet5, CapsuleNN, and AlexNet, and it is the consequence of a highly aggressive model compression: indeed, building E^{2}CNNs of higher cardinality demands significant compression of the initial model. Consequently, the accuracy drop experienced by the individual pruned CNN instances can be very high, with the resulting ensemble not being able to fully recover. In particular, the 8-E^{2}CNNs implementation of CapsuleNN is not even included in the results presented, because the compressed models obtained reach accuracy levels lower than 50%.

2.3.3 Robustness improvements

To proceed with the analysis of E^{2}CNNs, this section evaluates its robustness against memory errors. The plots illustrated in Figure 2.4 depict the accuracy achieved by single-instance CNN models as well as different E^{2}CNNs implementations for different SRAM voltage supply levels. The error rates corresponding to each evaluated voltage level can be retrieved from Table 2.3. The rightmost points, showing the accuracy achieved at 850 mV, assume no memory errors (i.e., nominal voltage level), while the leftmost points, corresponding to a voltage level of 650 mV, assume the highest evaluated error density (i.e., 2.2×10^{-3}). The obtained results demonstrate the higher resiliency against memory errors of E^{2}CNNs implementations with respect



(a) LeNet5 *(b)* AlexNet

(c) CapsuleNN *(d)* GoogLeNet

(e) MobileNet

Figure 2.4: Average accuracy for different error rates in the baseline single-instance CNNs (black) and in the corresponding E^{2}CNNs implementations comprising two (red), four (blue), and eight (green) instances.

to single-instance CNNs. Moreover, these results indicate that the higher the cardinality N of E^{2}CNNs, the higher its robustness at high error rates. Figure 2.4b clearly shows that the accuracy of the 8-E^{2}CNNs is significantly higher than the accuracy achieved by E^{2}CNNs of lower cardinality at 650 mV. Nevertheless, the 8-E^{2}CNNs performs worse when no errors affect the system, thus motivating the rationale behind the heuristic cardinality selector discussed in Section 2.2.3.

2.3.4 Reducing energy and memory requirements

In conclusion, I herein briefly present a potential use of E^{2}CNNs as a tool to reduce energy, as well as memory requirements. Previous sections have demonstrated how E^{2}CNNs is a more robust design solution than single-instance alternatives. Results have shown that its increased resiliency against memory errors can be exploited by allowing memories to operate at sub-nominal conditions to reduce energy. The energy reductions in the memory sub-system can be estimated by combining the results in Figure 2.4 with the energy numbers reported in Table 2.3. In particular, E^{2}CNNs implementations of LeNet5, AlexNet, CapsuleNN, and GoogLeNet can enable SRAM memories to operate at just 700 mV without affecting the baseline accuracy. As a result, the dynamic energy for read and write operations can be reduced by 28%. Similarly, MobileNet allows reductions of 20%, when the voltage supply is decreased to 750 mV.

However, E^{2}CNNs can enable more significant energy savings (and inference speed-ups) in specific, slightly oversized, benchmarks. In particular, if the accuracy of individual CNN instances is not significantly affected by pruning, the deployment of only M instances in a N-E^{2}CNNs, with $M < N$ could be possible. By reducing the number of instances in the ensemble, the runtime is reduced, as well as the inference energy and the memory requirements. For example, I have shown in [56] that deploying only 4 instances in an 8-E^{2}CNNs designs based on LeNet5, AlexNet, and GoogLeNet does not affect accuracy but improves energy, performance, and memory size by 2×. Finally, these experiments also show that this approach is completely orthogonal to the benefits obtained via voltage scaling: in the same work, my colleagues and I show how higher savings can be obtained by combining the

two strategies. For example, it is possible to deploy only four instances out of eight in an 8-E^{2}CNNs GoogLeNet-based implementation and reduce the SRAM supply-voltage to just 750 mV to obtain more than 53% energy savings without affecting accuracy.

3

Codebook-based compression

3.1 Introduction

One of the key motivations behind the proposal of E^{2}CNNs was that state-of-the-art stacking-based ensembling methods produce significant memory and computing overhead. These drawbacks are a well-known limiting factor for deploying most deep learning models in embedded devices, where resource constraints and low-energy budgets are key. Even more, the current shift towards more sustainable computing environments is making these concerns relevant in applications employing a cloud-computing approach. As a consequence, software-level optimizations that reduce complexity in deep learning models are becoming more and more employed in a large majority of scenarios. As discussed in Chapter 1, examples of popular model transformations targeting CNN models (but, more generally, most machine learning algorithms) include pruning [35], quantization [34], and encoding [37]. In this context, E^{2}CNNs can also be viewed as an algorithmic-level transformation that aims at similar goals, as I have illustrated in Chapter 2 how it can serve as a method to optimize memory needs. Nevertheless, in its original definition where all defined instances are instantiated, E^{2}CNNs transforms the input model to increase its error resiliency, but does not reduce memory or computing requirements.

Conversely, another avenue toward the compression of CNN models is that of codebook-based methods. These are classes of data representation trans-

formations that employ small look-up tables to store a limited number of unique representative values. As I will illustrate in detail in the following sections, compression can be achieved by leveraging the small size of codebooks and by transforming the typical value-oriented model representation into an index-based one. In contrast to quantization, where weight values are approximated using low-bitwidth formats, codebook-based strategies can achieve compression while including floating-point values inside codebooks, thus preserving high-precision floating-point arithmetic.

This chapter presents an effective and novel way to employ codebooks to compress CNN models. The strategy, that I have originally proposed in [78], is orthogonal to common compressing methods proposed in the literature for neural networks. Therefore, compression can be achieved without requiring ad hoc quantization schemes for HW acceleration and without reducing the number of model parameters by means of pruning methods. As such, the proposed codebook-based transformation can be employed in either safety-critical applications, to reduce memory requirements while preserving high QoS thanks to accurate floating-point arithmetic, or in edge applications, where it represents a model optimization that leaves the door open for further HW-SW co-design strategies, including, for example, quantization and encoding.

3.1.1 Codebooks-based representations

As the name suggests, codebook-based representations use codebooks as core elements for data representation. Codebooks are very tiny look-up tables that typically store the values or parameters used by a certain application. In general, codebooks usually have limited sizes, and their entries are generated using clustering methods. An example illustrating how codebooks are derived is shown in Figure 3.1, where the original input data is distributed in such a way that three different groups or classes can be individuated (green, blue, and yellow areas in the presented example). Clustering methods are machine learning algorithms that split input samples into different groups (or clusters), according to specific distance-based metrics. One typical implementation is represented by the k-means clustering, where input samples are grouped into k different clusters based on the Euclidean distance between each sample

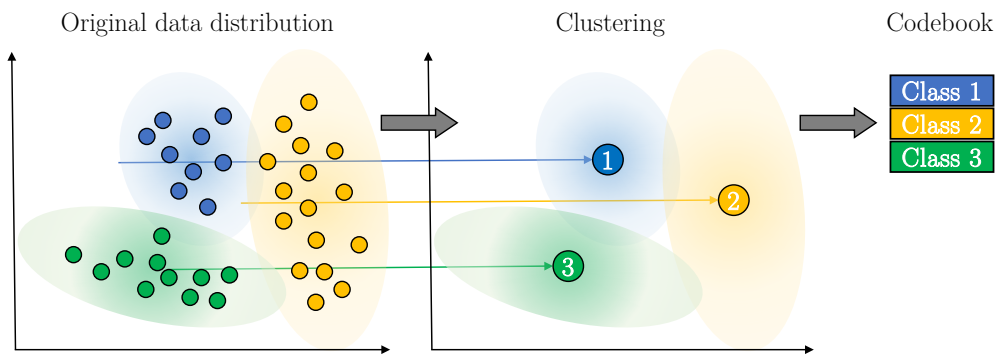


Figure 3.1: Codebook-based representations based on data clustering. The original floating-point data (left) is clustered in three separate groups (center). The floating-point centroid of each group is used as a representative value in the 3-entry codebook (right).

and k centroids which represent average values or features of each group. In the illustrated example, codebooks store the computed centroid of each cluster, thus reducing the number of inputs to only three distinct values (i.e., the three centroids).

When considering neural networks, codebooks can be used to store the weight and bias values of CNN layers. Indeed, in the context of CNN compression methods, codebooks serve as storage elements of a limited set of *unique* weight values that should well represent the distribution of the original weights. Clustering methods are hence applied to the input weights to reduce their number to *few* unique values, ultimately achieving CNN compression [39, 79]. The key insight behind this approach is that the set of samples belonging to a certain cluster can be approximated by the corresponding centroid. In this way, assuming that their individual values are substituted with the corresponding centroids, it would be possible to transform each weight from a floating-point number to an address of the generated codebook. Referring once more to the example shown in Figure 3.1, we can assume points as floating-point weights. In this case, each row of the codebook stores the floating-point value of one of the three centroids (i.e., the three representative weights, as determined by the clustering algorithm). Next, each weight can be converted to a 2-bit index, accessing the entry in the codebook corresponding to the associated centroid (i.e., 2-bit indexes

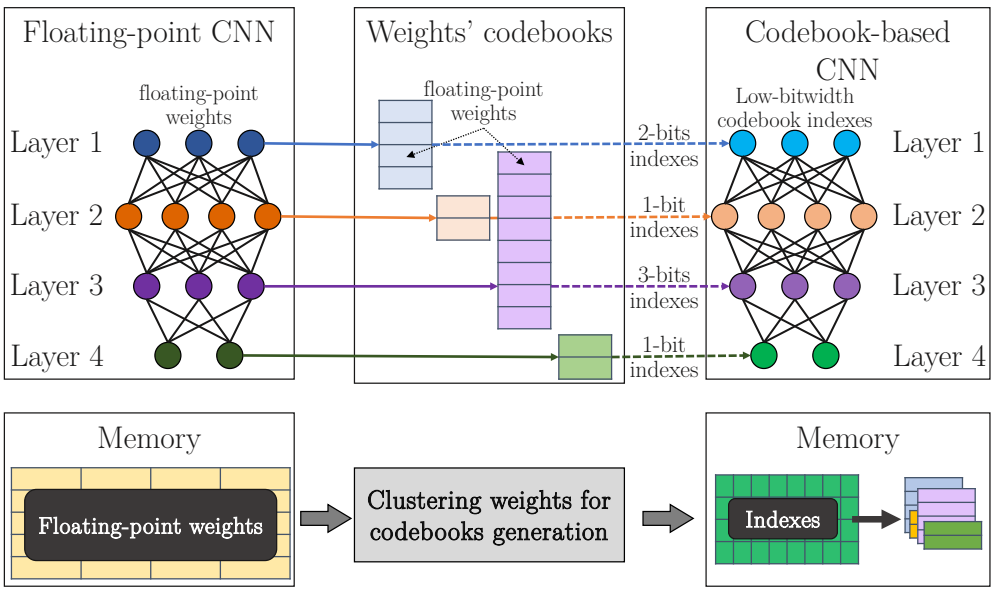


Figure 3.2: Using codebooks to compress CNNs. The floating-point weights in each layer (left) are clustered, and the computed centroids are stored in heterogeneously sized codebooks (center). The CNN weight values are then replaced by low-bitwidth indexes addressing the generated codebooks (right).

are enough to address up to 4-entry codebooks). In the next section, I will describe how this strategy can be effectively applied to CNN models in an accuracy-driven manner to reduce memory requirements.

3.1.2 Using codebooks to represent CNN models

Codebook-based representations can be used to compress CNN models. Section 3.1.1 has already presented the key insight to understanding how weight and bias values of convolutional and fully-connected layers can be represented using codebooks. The key idea is to move from a value-based representation of these layers into an index-based one. Figure 3.2 illustrates this process. The baseline model is a typical floating-point representation of a CNN (Figure 3.2-left). Then, a reduced number of weight and bias parameters determined via clustering algorithms are stored inside codebooks (Figure 3.2-center). In this example, this process is done on a layer basis, thus generating

multiple codebooks of different sizes. Finally, the CNN model is mapped as a set of indexes allowing the access of specific codebook entries to retrieve the corresponding floating-point weights (Figure 3.2-right).

Notice that simply storing inside the codebooks *all* the CNN parameters and then transforming the model as a set of codebooks indexes is not sufficient to get any storage benefits. On the contrary, it increases the memory requirements, as, in addition to weights and biases, an index for each parameter must be stored as well. On the opposite, significant memory reductions can be achieved when limiting the number of *unique* weight values in each layer. In fact, clustering weights into only k unique values requires relatively small codebooks that can then be accessed using low-bitwidth indexes.

As an example, let's imagine a CNN layer with 100 thousand parameters, a common size for convolutional layers in most recent CNN models. Assuming floating-point weights (i.e., 32-bit values), the storage of that layer requires:

$$M_{size} = W_{count} * W_{width} = 3,200,000 bits \tag{3.1}$$

with W_{count} being the number of weights, and W_{width} being the bitwidth of each weight.

Instead, by restricting the number of unique values to just 100, the generated codebook can be accessed using $\lceil log_2(100) \rceil = 7$ bits. Therefore, the storage requirements of the layer shrink to:

$$M_{size} = Codebook_{size} + W_{count} * idx_{width} = 712,800 bits \tag{3.2}$$

where $Codebook_{size}$ is the memory occupation of the codebook (i.e., number of entries × 32 bits), W_{count} is again the number of weights, and idx_{width} is the bitwidth of the codebook indexes replacing the original weight values

(i.e., 7 bits in this example). The result is that the original storage capacity is reduced by more than 4×.

While this approach can potentially enable significant memory savings, it must be considered that aggressively limiting the number of unique weight values (to obtain very low-bitwidth indexes) may introduce large approximations in the weight distribution, with a consequent impact on accuracy. In this regard, previous works on codebook-based methods for model compression have showcased memory reductions of up to 8×, with limited impact on output quality [37, 39]. Yet, they present major shortcomings. First, they adopt uniform approaches, in which all layers use the same number of clusters k, thus reducing a wide design space to a few (possibly sub-optimal) alternatives. Second, they do not include an accuracy constraint in their compression procedure. As discussed in Chapter 1, the impact on accuracy of different compression levels cannot be determined a priori. Therefore, state-of-the-art codebook-based compression strategies require a trial-and-error approach when having to abide by a user-defined accuracy threshold. The following section presents a methodology I proposed in [78] to solve this challenge.

3.2 Codebook-based compression methodology

In contrast to common codebook-based compression approaches, the proposed strategy employs non-uniform representations to better navigate through the complex trade-off between accuracy and compression. Moreover, previous works on codebook-based methods for CNN compression [33, 39] emphasize memory reductions at the cost of limited accuracy drop, but without controlling accuracy degradation. Instead, the approach presented in this section includes a user-defined accuracy constraint in the optimization loop, thus compressing the baseline model while abiding by a target accuracy level. In addition to model compression, Section 3.3.3 shows that this solution can also improve runtime performance. Finally, in contrast to previous proposals that either compress fully-connected [79] or convolutional [80] layers, the proposed approach targets both, hence allowing higher optimization possibilities in a more vast range of CNN models.

As previously discussed, the proposed methodology compresses CNN models only via a codebook-based approach, leaving for future works the possibility to analyze the effects of additional optimizations. For example, the authors of [37] described how orthogonal approaches such as quantization, Huffman coding, and pruning can be effectively used in conjunction with the use of codebooks to achieve even higher compression gains. However, the proposed methodology combines a codebook-based compression strategy with the E^2CNNs design described in Chapter 2. Although additional memory reductions could be achieved by reducing the number of deployed instances (see Section 2.3.4), E^2CNNs is only used to improve accuracy and robustness in this analysis. Thanks to this combined algorithmic-level optimization, the CNN compression strategy based on codebook-based representations is able to achieve compression levels higher than previous works for similar accuracy degradations.

3.2.1 Target: general purpose systems

I discussed in Chapter 1 the importance of a HW-SW co-design approach for the definition and deployment of CNN models in edge AI applications. Usually, this strategy demands using custom HW accelerators that trade off flexibility for higher computing efficiency, a key aspect for embedded systems. Recent research efforts addressed the issue of executing heavy workloads like CNN models into constrained edge devices from different perspectives, either providing edge devices with dedicated powerful and efficient hardware resources [81] or optimizing CNN models to reduce their complexity [82]. Usually, the two strategies are in fact combined together, with software-level optimizations trying to leverage the available hardware resources and capabilities. A common example of this co-design methodology is that of quantization, where the bitwidth of operands is adjusted to be efficiently used in the underlying specialized hardware, typically supporting integer arithmetic only. However, this approach may not be a viable solution under all circumstances. For certain applications, the use of floating-point arithmetic is a must, especially when executing safety-critical tasks. In addition, this solution requires the availability of ad-hoc hardware accelerators, which could not be accessible in some cases. In fact, the system can be usually defined a priori, hence forcing the application to adapt to the hardware resources.

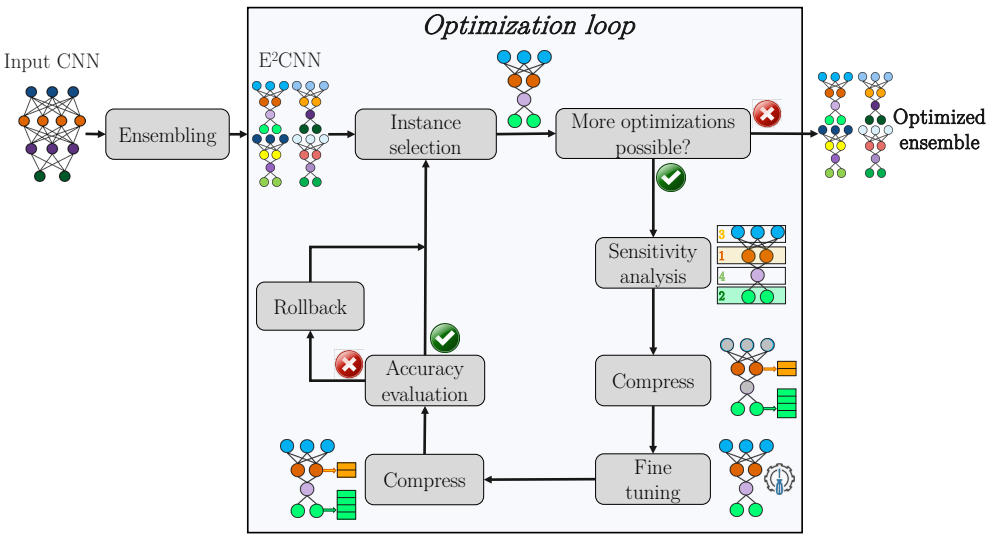


Figure 3.3: Compression methodology. The initial CNN is first transformed into an E^{2}CNNs equivalent to increase robustness. Then, a clustering-based optimization loop generates increasingly smaller codebooks on a layer basis to reduce memory requirements.

With these considerations in mind, the proposed codebook-based compression methodology enables potential compression and efficiency gains without the need for specialized, low-bitwidth hardware resources. In fact, codebook indexes of arbitrary bitwidth can be easily retrieved with standard mask and shift operations in any general-purpose computing system. Therefore, each memory word can accommodate multiple indexes, extracted at runtime, and employed to fetch the corresponding floating-point weights.

3.2.2 Heterogeneous codebook-based compression strategy

A schematic illustration of the proposed compression methodology is presented in Figure 3.3. It mainly consists of an optimization loop that iteratively reduces the size of codebooks in the layers of an E^{2}CNNs implementation, obtained, as a first step, by transforming an input CNN model. In the following, each stage of this approach is explained in detail.

3.2.2.1 Use of E^{2}CNNs in contrast to single-instance model

The input of the presented methodology is a single-instance CNN. As a first step, an equivalent ensemble model is derived to increase robustness (*Ensembling* in Figure 3.3), following the E^{2}CNNs methodology described in Chapter 2. As part of its construction stage, the obtained ensemble is trained before undergoing the optimization loop. Then, only one instance is selected and optimized at each iteration, but all of them are evaluated in this loop in a round-robin fashion. Doing so, a careful control of accuracy degradation is achieved and, by considering different CNN instances in consecutive optimization steps, the proposed strategy returns similar compression levels among the considered CNNs. Therefore, it prevents the resulting ensemble-based design from being composed of both highly-optimized and poorly-optimized CNNs, with a consequent improvement of the overall ensemble compression.

3.2.2.2 Per-layer iterative compression method

Compression is achieved by independently clustering weights and biases of each convolutional and fully-connected layer of a target ensemble instance. As a result, two codebooks are generated in each layer (i.e., one containing the weights and the other containing the biases). This choice allows a more accurate clustering of weights and biases and is motivated by the different distribution of these values. If a layer is selected as a possible target for optimization, the number of weight and bias values in its codebooks is halved. In this case, the bits required to index the corresponding codebooks are reduced by one. The way to select a layer (or multiple layers) at each step is described in the following section.

Note that only a subset of layers are targeted at each iteration, and a rollback is performed if the accuracy degradation threshold is violated. Hence, this strategy results in non-uniform codebook sizes, both in different layers of the same CNN instance (internal heterogeneity) and in the same layers belonging to different instances (external heterogeneity). The first optimization step always transforms a layer from its baseline value-based representation to a new codebook-based one. Codebooks with 32 entries, and thus 5-bit indexes (i.e., $K = 32$), are used as a first compression step, as empirical obser-

vations reported no accuracy loss for this compression level in all experiments described in Section 3.3. Then, the following optimization phases try to iteratively limit the number of codebook entries to just 16, 8, 4, or 2 values, corresponding to 4, 3, 2, or 1-bit indexes, respectively.

The generation (or, more generally, the reduction of the number of entries) of codebooks in the considered layer(s) is then followed by a few re-training epochs that try to compensate for the potential accuracy degradation. Nevertheless, re-training affects all the layers of the target CNN instance. Updating weight values breaks the original codebook-based representation employed in previously compressed layers. Therefore, to restore a proper codebook-based representation in those layers, a second clustering stage takes place after the re-training phase (*Compress - Fine tuning - Compress* in Figure 3.3).

3.2.2.3 Sensitivity-based logarithmic batch optimization

At each iteration, the methodology evaluates the robustness of convolutional and fully-connected layers of the target ensemble's instance. The goal is to determine which ones should be compressed to minimize the impact on accuracy. To this end, I introduced the *clustering sensitivity* metric S, which measures the range-normalized variance of weights. S is defined as:

$$S(l) = \frac{\sigma^2(W_l)}{max(W_l) - min(W_l)} \tag{3.3}$$

where $S(l)$ represents the clustering sensitivity of layer l and W_l is the corresponding set of weights. Layers showing a low sensitivity are considered first as candidates for compression. In fact, low S values indicate that weights are mostly concentrated in a small region of their entire range. As a consequence, clustering them does not introduce large absolute-magnitude deviations.

Instead of compressing a single layer at each optimization step, N layers, selected as the ones having the lowest sensitivity $S(l)$ values, can be compressed in parallel. In this way, the execution time of the proposed compression strategy is reduced. The value of N identifies the optimization batch size. Large

choices of N produce significant speed-ups of the optimization process, but introduce larger approximations that can extensively affect accuracy. On the other hand, small values of N better preserve accuracy at the cost of longer execution times.

The compression stage, implemented as the compress-retrain-compress procedure detailed above, is then followed by an accuracy evaluation. In particular, the accuracy of E^{2}CNNs, instead of the individual instance currently under optimization, is evaluated. If the accuracy meets the user-defined accuracy constraint, the optimization is retained and the algorithm continues. A new iteration starts, another CNN instance is selected, and a new batch of layers becomes the target for the optimization step. Instead, if the accuracy degradation exceeds the threshold, the performed optimization is discarded, the previous instance implementation is restored, and, for the targeted CNN instance only, the batch size N is halved. Halving the batch size is a key component of the proposed methodology, as it allows for a more fine-grained approximation in a future stage when the same instance is selected again. When $N = 1$ for a CNN instance (i.e., a batch size of 1), no further optimization is considered for the targeted layer when a compression iteration produces an unacceptable accuracy. Finally, the compression procedure ends when no layer can be selected for further optimization.

3.2.2.4 Complexity analysis

The presented compression strategy has a complexity linear in the number of instances I of E^{2}CNNs, as well as in the number C of considered compression levels. In particular, assuming K being the highest number of possible entries in the generated codebooks, $C = log_2 K$, since the codebook size is halved at each iteration if failing in preserving accuracy. In the worst case, all batch clustering steps fail in compressing multiple layers in one single shot while meeting the accuracy constraint at the same time. When this happens, the batch size N always reaches 1, meaning that each layer is considered in isolation. In this case, the complexity becomes linear also in the number L of layers of each CNN, resulting in $\mathcal{O}(ICL)$.

Nevertheless, the proposed sensitivity-based layers selection and a relatively large initial batch size N support effective runtime reductions. In the best case, the model can be effectively compressed, preserving the initial batch size in all iterations, and reducing complexity to $\mathcal{O}(\frac{ICL}{N})$. Finally, the user-defined accuracy threshold plays an important role in this analysis, since very low acceptable accuracy drops (i.e., less than 1%) reduce the likelihood of successful compressions of a large batch of layers in parallel. Hence, multiple batch size reductions affect the runtime of the optimization algorithm from early iterations.

3.3 Experimental results

3.3.1 Experimental Set-up

The proposed compression methodology has been evaluated on the following benchmarks: LeNet5 [70], AlexNet [72], VGG16 [83], GoogLeNet [74], ResNeXt [84], and MobileNet [76], all trained and tested on the CIFAR100 dataset [73]. As in Chapter 2, the benchmarks considered in this evaluation differ in size and complexity to demonstrate the applicability of the proposed approach in a wide range of AI applications. As part of the first optimization stage, E^{2}CNNs implementations of different cardinality ($I = \{2,4,8\}$) are considered. For clarity, only configurations that maximize classification accuracy are presented in Section 3.3.2.

Accuracy evaluations are performed in PyTorch [85]. All benchmarks are trained for 200 epochs, using the same training hyper-parameters: the Adam optimizer, a fixed learning rate of 10^{-3}, and a weight decay of 10^{-4}. A deeper investigation to determine the best training configuration for each benchmark is not performed because out of the scope of this analysis. However, the baseline accuracies obtained are in line with those reported in the literature.

Compression is achieved using the k-means clustering function provided by Scikit-learn [86], using an initial uniform clusters distribution. As the authors of [37] demonstrate in their work, this approach better protects the initial classification accuracy by preserving high-magnitude parameters. The number of fine tuning epochs executed in each clustering-based compression

phase is set to 10, and it is followed by a second execution of the k-means algorithm that clusters again the weight values to their pre-training codebook-based representation. The batch size N is empirically set equal to half of the total number of layers. This choice results in a good compromise between compression and execution time. In addition to the aforementioned PyTorch implementation, a C++ inference solver (based on the same framework described in Chapter 2) is adopted to measure inference runtime performance. To accurately simulate the behavior of the compressed benchmarks, the index-based weight accessing is implemented, thus using shift-mask operations to retrieve codebook indexes and ultimately get the clustered weight values.

Two state-of-the-art baselines are used to compare the compression/accuracy trade-off achieved by the proposed methodology. The first one is uniform quantization, which represents a more traditional way to compress CNN models. The second baseline employs instead codebooks. In contrast to the proposed solution, this baseline forces codebooks to have the same size in all layers. This approach is similar to the one described in [33], but without the constraint of having symmetric weights and instead employing a layer-based incremental strategy. In addition, no accuracy control is performed in that work.

3.3.2 Compression/Accuracy trade-off

Before analyzing the accuracy/compression trade-off achieved with state-of-the-art approaches and with the proposed methodology, I present in Table 3.1 the accuracy improvements of E^{2}CNNs implementations with respect to their single-instance baselines. The achieved accuracy improvements of up to more than 5% confirm the findings I have illustrated in [56] and reported in Chapter 2, hence motivating the use of E^{2}CNNs as a preliminary model transformation in this methodology.

The comparison between the proposed codebook-based compression methodology with state-of-the-art alternatives is instead summarized in Figure 3.4. The six plots compare uniform quantization (black lines) and uniform codebook-based compression methods (red lines) with the

Table 3.1: Accuracy of single-instance and E^{2}CNNs implementations evaluated on the CIFAR100 dataset.

Benchmark	Accuracy		Accuracy Gain
	Single Inst.	E^{2}CNNs	
LeNet5	42.56%	43.63%	1.07%
AlexNet	55.84%	61.16%	5.32%
VGG16	69.24%	74.18%	4.94%
GoogLeNet	72.62%	76.48%	3.86%
ResNeXt	75.33%	78.09%	2.76%
MobileNet	64.61%	67.74%	3.13%

proposed solution considering three different accuracy thresholds of 0.5% (green triangles), 1% (yellow triangles), and 5% (blue triangles).

These results indicate that, from a model compression perspective, codebook-based strategies outperform quantization, providing higher accuracy levels for the same compression. In the case of quantization, compression refers to the bitwidths used to represent weight values, while in the case of codebook-based compression, it refers to the reduced size of codebook indexes. Considering optimized models reaching accuracy within 1% of their baselines, the uniform codebook-based strategy enables more aggressive bitwidth reductions, achieving an average 29.16% higher compression when compared to quantization. The superior accuracy of the codebook-based approach derives from its higher flexibility, as the selected weight values stored inside codebooks better adapt to the original distribution during the clustering process. In fact, weight values obtained using a uniform quantization approach span the representable range *uniformly*, with any pair of two consecutive values being separated by a fixed amount (i.e., the quantization step, or resolution). Conversely, clustering does not limit the selected centroids to assume equally-distant values, thus more accurately representing the input weights distribution.

Anyway, the two presented solutions do not control accuracy. Therefore, sudden accuracy drops affect the optimized models when the size of all codebooks is aggressively reduced beyond a critical point that, as previously

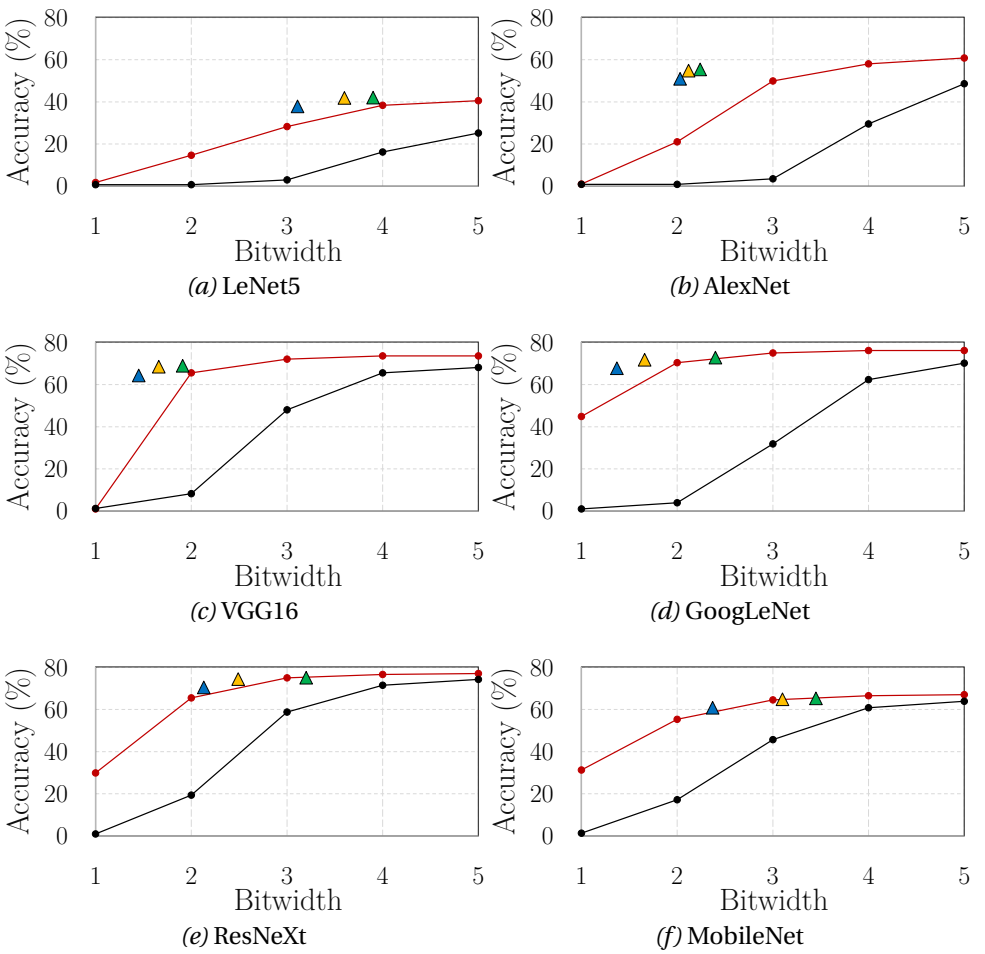


Figure 3.4: Accuracy achieved in compressed benchmarks. Uniform quantization (black) and uniform codebook-based compression (red) correspond to previous state-of-the-art works. Green, yellow, and blue triangles refer to the proposed strategy considering 0.5%, 1%, and 5% accuracy thresholds, respectively.

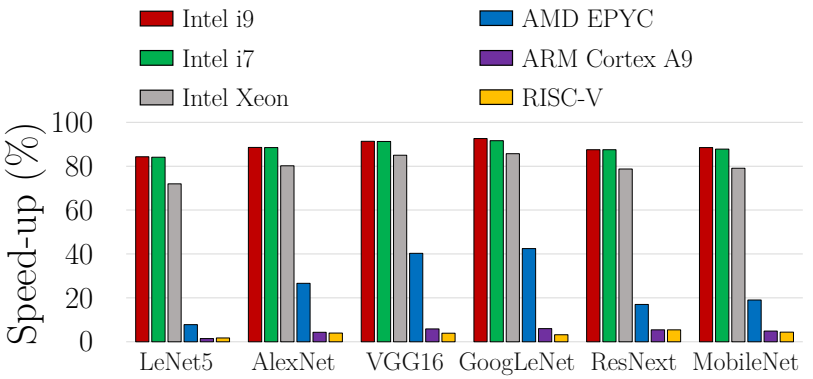


Figure 3.5: Performance gains in different processors

discussed, cannot be determined in advance. Conversely, the proposed strategy addresses this issue by introducing an accuracy-driven optimization loop. While the ability to preserve accuracy to user-defined levels is already an improvement with respect to state-of-the-art alternatives, the proposed solution also offers more favorable accuracy/compression trade-offs, due to its heterogeneous nature and thanks to the higher robustness of E^{2}CNNs designs. Indeed, Figure 3.4 shows that the described methodology improves compression across all benchmarks by an average of 22.27%, 8.31%, and 13.13% for accuracy thresholds of 0.5%, 1%, and 5%, respectively, when compared to uniformly codebook-based compressed alternatives. Moreover, considering single-instance floating-point baselines, this approach enables average compressions of 11×, 13×, and 15×, respectively, for the three evaluated accuracy thresholds. Improvements in the considered trade-off derive from adopting different compression schemes in the CNN layers. Thus, deeper compressions (i.e., smaller codebooks) are only applied in layers exhibiting a higher degree of robustness.

3.3.3 Performance gains

It would be reasonable to expect a performance slow-down when executing the benchmarks compressed using the proposed codebook-based strategy. Indeed, masking and shifting operations are required to retrieve an index of arbitrary bitwidth from a word (usually composed of 32 or 64 bits) read from memory. For example, to get a 6-bit codebook index from a 32-bit mem-

ory word, a mask operation is first required to discard 26 bits. This can be achieved with a logical *AND* operation with the 32-bit memory word containing the desired index and a mask with *1s* in the bit positions corresponding to the index, and *0s* everywhere else. Then, the memory word must be (right)-shifted to move the 6-bit index to the 6 LSBs. The index is then retrieved, but an additional level of indirection is introduced, as this index is used as an address in a look-up table (i.e., the codebook) storing the corresponding floating-point weight value.

Counter-intuitively, the presented methodology offers instead a significant *reduction* in inference runtime. The time required for inferences using the baseline floating-point implementation of the benchmarks is measured and compared to those optimized with the proposed methodology, considering an accuracy degradation threshold of 1%. Figure 3.5 illustrates the achieved results for five different systems-on-chip of different computing power: SiFive Freedom U740 (based on the RISC-V ISA), ARM Cortex A9 MPCore, AMD EPYC 7551, Intel Xeon Gold 6154, Intel i7-4790, and Intel i9-10900K. The measured speed-ups are strongly correlated to the reduction of memory accesses in the optimized benchmarks: on one side, it decreases the energy-per-inference and, on the other side, it reduces the probability of cache misses, with clear performance benefits. Codebooks are usually small and, therefore, are retrieved from local cache memories. Moreover, thanks to their reduced bitwidth, multiple indexes are read at each word access in the main memory, thus reducing the number of (slow) read operations. Improvements are indeed more substantial in high-performance CPUs, for which the clock-cycle cost of memory accesses is far more expensive than the one required for arithmetic operations (e.g., multiplications and additions in MACs). Therefore, the achieved reduction of memory operations limits the number of cache misses, thus producing very high speedups. Average improvements range from less than 5% on RISC-V and ARM processors, to more than 80% on high-performance Intel CPUs. Our results also suggest that the overhead considered of shift and mask operations required to extract the bit fields corresponding to low-bitwidth indexes from memory words, as well as the time required to retrieve floating-point weights from the codebooks, are negligible with respect to memory read and write operations.

4

Approximate Computing

4.1 Introduction

In Chapter 1, I have explained how deeply the edge computing paradigm [30] is revolutionizing the field of Artificial Intelligence (AI), impacting scenarios ranging from personalized healthcare to astronomy [9, 11, 15, 17, 19]. Discussing the benefits of this computing strategy, I focused on Convolutional Neural Networks (CNNs) as extensively investigated models, with several optimization efforts proposed by the research community. Their increasing complexity translates into very compute-intense workloads, demanding the execution of millions of multiply-accumulate (MAC) operations. This trend, strains the capabilities of ultra-low-power embedded systems, especially in the case of edge computing. Two widely employed optimization avenues include pruning approaches [35], which remove specific neural connections or entire computational blocks from CNN models, and quantization strategies [33], that instead reduce the bitwidth of CNN operands. In Chapter 3, I have presented codebook-based representation as another approach to reduce the memory requirements of CNN models, proposing a methodology to compress CNN models (or the E^2CNNs implementations discussed in Chapter 2) targeting general-purpose computing devices, that do not necessarily employ specialized hardware resources. That approach targets the case of architectural designs where the computing system is already defined, thus not allowing the introduction of custom HW accelerators.

Nonetheless, I have also underlined that, in general, a hardware-software (HW-SW) co-design approach is key to achieving important gains in computation efficiency. Herein, I analyze possible HW-SW co-design strategies to improve energy efficiency in embedded systems. Indeed, another way to optimize the performance of CNNs is the use of approximate operators that trade arithmetic correctness for efficiency [87]. The implementation of hardware units producing approximated arithmetical results can be beneficial to reduce energy in AI applications. Since models like CNNs can tolerate a certain degree of noise, the exact multiplier and adder units can be effectively replaced by inexact counterparts, without significantly affecting their output quality. The main advantage is that inexact arithmetic operators are specifically optimized from area, energy, and latency perspectives, ultimately allowing the control of the trade-off between introduced approximation and efficiency improvements.

Focusing on inexact multipliers, I propose a two-stage optimization methodology where inexact arithmetic is employed in carefully selected layers of CNN models to increase their inference efficiency while abiding by user-defined accuracy constraints. The optimization strategy combines quantization and inexact arithmetic with a sensitivity analysis that evaluates the robustness of individual CNN layers to select the ones that are robust enough to be executed using an approximate multiplier. This approach has been presented in [88].

4.1.1 Approximate computing

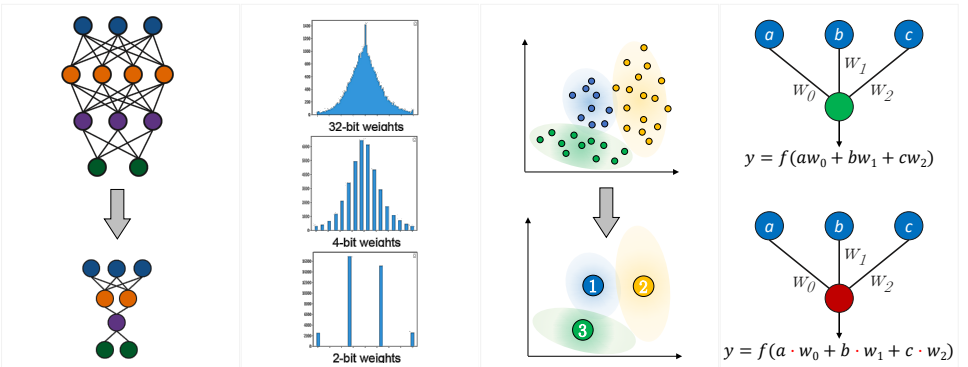
In general terms, the approximate computing paradigm encompasses strategies trading off the exactness of computed outputs with performance metrics such as runtime and energy consumption [89]. Approaches and studies related to Approximate Logic Synthesis (ALS) are of particular relevance in this regard, because they can derive inexact, yet extremely efficient, arithmetic circuits for widely used operators [90]. These can then be employed as building blocks for more complex hardware accelerators [91].

Adders represent one of the most investigated classes of arithmetic units as they are largely employed in most applications and workloads, and also

because they are the fundamental building blocks for more complex units (e.g., they can be combined to implement multipliers). In inexact operators, the arithmetic approximation is tuned at design time and typically introduced at the gate level. For example, XOR gates are required to compute the *sum* bit in half adders. However, replacing the XOR gate with an OR gate introduces an error in a single row of the corresponding 4-entry truth table. Indeed, when both inputs are '1', the XOR gate returns '0', while the OR gate outputs '1'. For the other three combinations, both implementations produce the same result. Still, the OR implementation is a much better option in terms of area, latency, and energy, thus being a good candidate to leverage the arithmetic inexactness to improve efficiency. As a result, approximate operators such as adders and multipliers can be designed using this gate-level approximation to construct inexact alternatives [50].

4.1.2 Approximations in machine learning

Thanks to their intrinsic redundancy and robustness, machine learning models, and CNNs in particular, have been extensively studied in conjunction to approximate computing methods. In a sense, most of the techniques discussed in the previous chapters can be seen as forms of approximate computing. A schematic representation of the different optimization methods that introduce approximation into CNN models is summarized in Figure 4.1. Removing convolutional filters from a baseline CNN model (i.e., pruning) is indeed a form of approximation, as the objective is to obtain a simplified architecture that, compared to the input network, produces *similar* output qualities, but exhibits a lower number of parameters. The approximation introduced by quantization is an even more standard way to consider approximate computing in CNN models. In fact, reducing the arithmetic precision of weights and activations means approximating inputs and outputs with respect to the floating-point values. Weight clustering introduces inexactness in a way similar to quantization. Indeed, clustering weight values and replacing them using the computed centroids is another form of approximation. Finally, approximate operators are circuits of high interest in the machine learning community. Among the plethora of approximate operators, this chapter restricts its focus to multipliers, mainly due to their relatively high energy footprint (e.g., with respect to adders). Moreover, as the main bench-



(a) Filter pruning *(b)* Quantization *(c)* Clustering *(d)* Operators

Figure 4.1: Common optimization strategies for CNNs usually introduce a specific form of approximation. Pruning reduces computing complexity (a), while quantization and clustering approximate data values (b, c). Finally, approximate operators implement inexact arithmetic (d).

marks evaluated in this thesis are neural network models, multipliers result in highly stressed units when executing these workloads and can hence significantly contribute to the overall energy consumption of an edge computing node.

4.1.3 Approximate multipliers

Approximate multipliers are of particular interest when designing approximate arithmetic units for CNN accelerators, as they usually exhibit parallel and compute-intensive structures, where a major contribution to resource and energy budgets is the arithmetic logic in their datapaths [92, 93]. The authors of [55, 94] highlight that, when considering CNNs, multipliers are the most amenable target for approximation. In fact, multipliers typically present a high energy footprint and are largely used in neural networks that require a very high number of multiplications to run inference. ALS methods can be used to design approximate operators of specific approximation degrees. For example, these methods are used by the authors of [50] to derive a multitude of adders and multipliers which differ in their physical characteristics as well as in the inexactness degree of their outputs.

Instead of focusing on approximate multipliers design for CNN efficient executions, this chapter presents an application-mapping methodology, aiming to leverage the available energy-saving opportunities in inexact hardware resources while controlling accuracy degradation. Similarly to the codebook-based compression methodology discussed in Chapter 3, the impact of inexact arithmetic on CNNs output quality cannot be evaluated at design time, as the effects also depend on the CNN structure and on the application complexity. For example, the authors of [55] analyzed the impact of different inexact multipliers on the convolutional and fully-connected layers of VGG16 and concluded that the first and last layers are particularly sensitive to approximation. Based on this observation, they suggest a hybrid approach where only the central layers are executed using approximate multipliers. A similar observation is presented in [95], although the authors of this work investigate the higher sensitivity of the first and last layers towards analog noise. Nonetheless, this chapter will show how the finding of [55] does not hold in general, when applied to diverse CNN models.

4.2 Unleash inexact arithmetic in CNNs

This section shows that the judicious use of approximate multipliers in highly optimized (quantized) models can further improve efficiency beyond that attainable by quantization alone. Hence, the proposed approach carefully maps them to execute specific CNN layers and combines them with orthogonal state-of-the-art CNN optimization strategies to fully exploit their benefits. To guide the selection of CNN layers where inexact arithmetic can be applied while abiding by a certain user-defined accuracy level, I propose a heuristic optimization strategy that evaluates the resiliency of individual layers and that aims at improving efficiency while limiting the search space to reduce the complexity and runtime of the proposed methodology. This approach logically separates the approximation degree of the employed inexact multiplier from the target accuracy level to achieve, making these two values independent input parameters in the developed methodology. Like other previous works analyzed in previous chapters, I have noticed that studies on approximate multipliers for CNN models do not explicitly control accuracy [55]. Conversely, I consider the accuracy-driven nature of the proposed optimiza-

tion strategy a key element in improving state-of-the-art proposals. Therefore, I present an incremental mapping strategy, where accuracy is effectively controlled at each optimization step, independently of the approximation degree of the multiplier itself.

As also done for the codebook-based compression methodology presented in Chapter 3, I transform the input single-instance CNN into an E^{2}CNNs equivalent to improve its robustness. Then, I combine the use of inexact arithmetic with a tailored per-layer quantization and show how the combination of the two methods can be effectively employed to fully leverage the available hardware resources to reduce energy.

4.2.1 Methodology overview: a co-design vision

The methodology discussed in this section combines algorithmic and hardware optimizations into a single design approach. On the algorithmic side, the E^{2}CNNs methodology presented in Chapter 2 and a HW-aware per-layer quantization scheme are employed. Concerning E^{2}CNNs, the preliminary hypothesis is that its higher robustness against memory errors can be also leveraged in the presence of arithmetic approximations, thus allowing for more extensive uses of inexact resources while achieving a certain accuracy level. Experiments presented in Section 4.3 confirm this intuition. Instead, the proposed quantization strategy is oriented toward the available hardware elements: it considers the available multipliers to determine acceptable quantization levels. An overview of the overall methodology is depicted in Figure 4.2.

4.2.2 Heterogeneous per-layer quantization

The methodology summarized in Figure 4.2 aims at effectively exploring the wide space of candidate designs that can result from different combinations of ensembling methods, heterogeneous quantization, and inexact operators. The optimization methodology accepts as input a single-instance CNN. First, it transforms the baseline model into an E^{2}CNNs equivalent to increase robustness (and accuracy). Then, it applies a per-layer quantization to reduce the use of memory bandwidth and computational resources in each layer. An

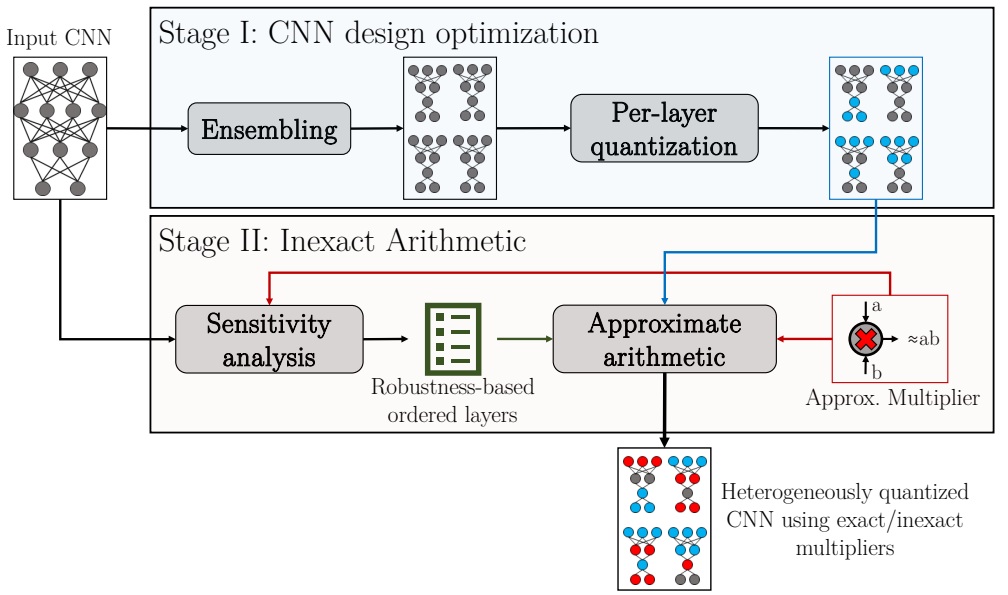


Figure 4.2: The proposed two-stage methodology. It combines together E^{2}CNNs, heterogeneous quantization strategies, and approximate computing to improve inference efficiency.

analysis of the layers of the baseline CNN model estimates their sensitivity to approximation due to a target approximate operator. Finally, it leverages the outcomes of the algorithmic-level optimizations and the results of the sensitivity analysis to map the quantized E^{2}CNNs design on approximate hardware resources, exploiting their lower power consumption to increase inference efficiency. These steps are implemented offline in two distinct stages.

4.2.2.1 Stage A: Robustness-Aware CNN Optimization

The input single-instance floating-point model is optimized in a two-step process. First, the structure of the input model is transformed into an E^{2}CNNs design to improve accuracy and robustness against data perturbations (*Ensembling* in Figure 4.2). E^{2}CNNs is derived by combining filter pruning and replication following the methodology presented in [56] and discussed in Chapter 2. Then, a uniform quantization scheme employing 8-bit weights and 16-bit activations (i.e., an 8/16 quantization scheme) is applied to all

the obtained CNN instances during the last training epochs. Previous experiments and studies have already shown that such a quantization level does not significantly reduce the baseline floating-point accuracy (see results in Chapter 2 or [54, 56]).

The generated ensemble is then further optimized with a second quantization step (*Per-layer quantization* in Figure 4.2). This phase targets each instance individually and proceeds per layer in topological order, reducing the bitwidth of weights and activations to 4 and 8 bits, respectively (i.e., a 4/8 quantization scheme). The 4/8 quantization level is applied to a certain layer only if the resulting accuracy abides by a user-defined constraint. Otherwise, the previous 8/16 quantization level is retained. When all layers (of all CNN instances) have been considered in the quantization optimization loop this procedure ends. As introduced before, CNN ensembling and per-layer quantization are employed in a combined approach. On one hand, the high robustness of E^2CNNs supports a more intense use of approximation, either implemented in the form of inexact arithmetic or in the form of quantization. On the other hand, E^2CNNs also enables a more fine-grained heterogeneous quantization, because of the two degrees of heterogeneity obtained thanks to the multi-layer optimization (internal heterogeneity) in a multi-instance architecture (external heterogeneity), as presented in Chapter 3. Moreover, the more compact quantization scheme applied to specific layers also reduces memory size and computing requirements. More importantly, it also improves efficiency by enabling the use of simpler, and therefore more efficient, multipliers for the execution of 4/8 quantized layers.

4.2.2.2 Stage B: Mapping on Inexact HW Resources

When executing a certain subset of CNN layers using a target approximate multiplier, the impact on accuracy is unpredictable at design time. The result is that a cautious mapping strategy is required to prevent undesirable output quality degradations. Therefore, in the second stage of the proposed methodology, a heuristic approach is used to select the layers robust enough to be implemented using inexact arithmetic. Only the layers quantized using the 8/16 scheme are considered candidate layers for the use of approximate arithmetic (i.e., the layers quantized more aggressively in Stage A are executed

with exact multipliers). First, the heuristic orders the layers according to their approximation sensitivity (*Sensitivity analysis* in Figure 4.2). Sensitivity is measured by instantiating a target approximate multiplier in only one layer of the single-instance CNN at a time to evaluate the resulting inference accuracy. This analysis is performed on the initial single-instance baseline, uniformly quantized on the 8/16 scheme. The results of this analysis are then extended to the CNN instances that compose the generated E^{2}CNNs design.

First, I have tested the same sensitivity analysis on each CNN instance of different E^{2}CNNs implementations and found that very similar outputs are produced. Hence, these results indicate that layers' resiliency may be more closely associated with their size and structure than with their actual weight values. Then, such an approach reduces the runtime execution of the heuristic method and optimization loop as the sensitivity analysis can be executed only once. Potentially, this stage can be executed in parallel to stage A, as it operates on the input single-instance model. Moreover, in contrast to an impractical exhaustive exploration, this heuristic can efficiently scale to large CNN applications. Indeed, being L the number of convolutional and fully-connected layers, the complexity of the presented methodology is linear in L, and can be thus expressed as $\mathcal{O}(L)$. In fact, the heterogeneous quantization optimization and the analysis to map CNN layers onto inexact multipliers are themselves of linear complexity. Therefore, applying them in sequence makes the entire procedure linear as well. The low complexity degree of the proposed heuristic methods enables a relatively fast design step, although it cannot guarantee that the optimal mapping configuration is found for specific accuracy thresholds. Such a solution can be only provided by an exhaustive exploration that would need to enumerate all the possible combinations (and test them). Thus, its complexity would be $\mathcal{O}(3^L)$, since 3 alternative implementations exist for each layer (i.e., 4/8 quantization, 8/16 quantization, 8/16 quantization with inexact multipliers), preventing its applicability in almost any practical application.

The choice of introducing approximation only in 8/16 quantized layers is well motivated by the results presented in Section 4.3. Still, preliminary tests on multiple 4/8 approximate multipliers available in [50] show that they all have a very large impact on accuracy when compared to their exact alternative.

Furthermore, the energy savings due to the use of an *exact* multiplier for 4/8 quantized layers is significantly higher than the one obtained by using *inexact* multipliers in 8/16 ones. Therefore, the proposed strategy allows the use of an approximate multiplier in 8/16 quantized layers only, using an exact implementation in the other layers. From the observation of larger savings from quantization than from approximate arithmetic derives also the choice of applying the heterogeneous quantization stage before the injection of arithmetic approximation. In fact, the two steps are executed using the full user-defined accuracy margin. Therefore, the initial quantization phase maximizes the number of layers quantized more aggressively, at the cost of leaving little or no additional accuracy degradation to trade off when introducing inexact arithmetic. This greedy approach is also motivated by a second observation showing how the use of certain approximate multipliers in individual layers may not even affect the output quality of the model (although a more extensive use definitely does it).

The results of the sensitivity analysis are then combined with the optimized ensemble, to iteratively introduce approximate multipliers in specific layers of the CNN instances of E^2CNNs, starting from the least sensitive ones. This phase terminates when no further layer can be approximated while complying with the accuracy constraint.

4.3 Results

4.3.1 Experimental Setup

Similar to the experiments presented in previous chapters, multiple CNN benchmarks are considered to evaluate the benefits of the proposed optimization strategy. For this analysis, the following models are used: AlexNet [72], VGG16 [83], GoogLeNet [74], ResNext [84], and MobileNet [76], all evaluated considering their Top-1 classification accuracy on the CIFAR-100 dataset [73].

These models are trained in PyTorch [85], using *fake quantization* functions as presented in [77] for the last 20 training epochs. Fake quantization preserves floating-point arithmetic, tuning weights and input activations of convolutional layers according to the desired quantization scheme. In this

way, these layers are still executed using floating-point arithmetic, but input operands are adjusted forcing them to assume values representable in the target quantization level. Therefore, this approach simulates quantization in the training process, but does not exactly reflect real fixed-point arithmetic. For this reason, accuracy evaluations are performed using a C++ inference solver, as discussed in Section 2.3.1.

As in [56], E^2CNNs designs containing 2, 4, or 8 instances are evaluated, presenting the results of the configuration achieving the highest accuracy. Efficiency is measured as the energy required by all exact and inexact multiplications executed in a CNN inference. The overall energy impact of MAC operations at the chip level is architecture-dependent: on one side, it is usually relatively low in single-core platforms where data movements account for the largest fraction of energy consumption. On the other side, it can dominate in multi-core edge AI accelerators comprising hundreds of processing elements [96], thus justifying this methodology.

Compared to the evaluation method implemented in [48], where inexactness is achieved by reducing operands' precision (i.e., similar to the effects of quantization), this study emulates the behavior of a (possibly inexact) target multiplier. To do so, I extend the developed C++ inference solver by replacing all multiply instructions with custom multiply implementations emulating the target inexact multiplier. The C++ code serving this purpose is publicly available [50].

Additionally, a second difference with respect to [55] is that the authors of that work apply approximation to float16 arithmetic by using approximation matrixes that simulate inexact operators. Instead, two distinct integer approximate multipliers are presented here. As in [88], I present an analysis showing the accuracy/energy results achieved employing two inexact multipliers selected from [50]. The selected multipliers differ in the approximation degree and serve to illustrate the applicability, effectiveness, and hardware-agnostic characteristics of the proposed methodology. In particular, the structure of the considered approximate multipliers is provided by [50] in the form of a Verilog description and is derived by employing a multi-objective Cartesian genetic programming approach (CGP), using various exact implementations

as baselines. Among the large number of potential candidates provided by this library, the two presented in this thesis have a significant approximation degree, which allows me to present two considerations: first, their significant arithmetic error mandates a precise mapping methodology since their use in multiple layers can critically impact the CNN outputs. As I will show later on in this section, using one of the two approximate multipliers to run *all* CNN layers has a dramatic effect on accuracy. Second, although their highly inexact nature results in significant energy savings (i.e., compared to exact implementations or other inexact alternatives), I will show that the implementation of an *exact* 4/8 multiplier (i.e., an exact implementation designed to operate with inputs of lower bitwidth) still enables more significant energy reductions.

To perform a fair comparison, the structure of the employed inexact multipliers is adapted to match the bitwidth of input and output operands in the quantized layers of the CNN models considered in this study: indeed, the original 16-bit multipliers defined in [50] are over-dimensioned for the 8/16 layers of the considered benchmarks, as, in the experiments I discuss, one input operand (i.e., the weight) requires only 8 bits. Therefore, the original Verilog implementation is modified, adjusting the bitwidth of input and output operands, as well as the bitwidth of the connected internal components. I characterized the power consumption of the circuits using Synopsys Design Compiler, employing HVT cells from the 40LP TSMC technology library (40nm, low power). The error induced by approximation is measured in terms of Mean Relative Error (MRE) by running a C++ simulation over all the possible input combinations. The synthesis and simulation results of the employed multipliers are summarized in Table 4.1. Exact16 and Exact8 are exact multipliers used in 8/16 and 4/8 layers, respectively, while MulF6B and Mul8VH are the target approximate multipliers used only in 8/16 layers.

As convolutional and fully-connected layers account for the largest percentage of MACs in the majority of recent CNN architectures, the proposed approach supports the use of arithmetic approximation in both types of layers (as opposed to [94], where only convolutional ones are candidate layers for approximation). To validate the methodology presented in Section 4.2, a system featuring two exact multiplier implementations (i.e., Exact16 and

Table 4.1: Operands bitwidth, mean relative error (MRE), and area/power characterization of the multipliers used in the proposed experiments.

Multiplier	Bitwidth	MRE (%)	Power (μW)	Area μm^2
Exact16	(8 × 16)	N/A	277.5	622.5
Exact8	(4 × 8)	N/A	39.9	94.8
MulF6B	(8 × 16)	5.9×10^{-5}	237.3	441.7
Mul8VH	(8 × 16)	1.9×10^{-3}	137.3	192.9

Exact8) and a single approximate multiplier (i.e., either MulF6B or Mul8VH) is considered in the following experiments.

4.3.2 Sensitivity analysis

Before focusing on the accuracy, energy, and area results achieved by the proposed optimization methodology, I will first show the results of the sensitivity analysis (discussed in Section 4.2) used to estimate the resiliency of individual layers of a CNN model against arithmetic approximation. This analysis is conducted using the Mul8VH inexact multiplier, since its higher approximation degree results in more significant accuracy degradations, ultimately better illustrating the intended considerations. A summary of this investigation is presented in Figure 4.3. The obtained results demonstrate that the measured resiliency is highly layer- and model-dependent. First, these results indicate that the use of a certain inexact multiplier (i.e., Mul8VH in this case) to run a specific layer of a CNN model affects accuracy in a way that varies significantly across different benchmarks. Second, these results also highlight that the obtained accuracy drop depends on the CNN structure. For example, the drop never exceeds 5% in VGG16. On the contrary, it can be very close to 10% in multiple layers of MobileNet and around 20% in AlexNet, on average. In particular, the high drop shown by this latter model can be due to the absence of normalization layers, which demonstrate to be highly effective in mitigating this behavior[1]. Finally, the impossibility of predicting the resiliency of CNN layers at design time makes the sensitivity analysis a

[1]Normalization layers are instead included after convolutional layers in all the other CNN models considered in this analysis.

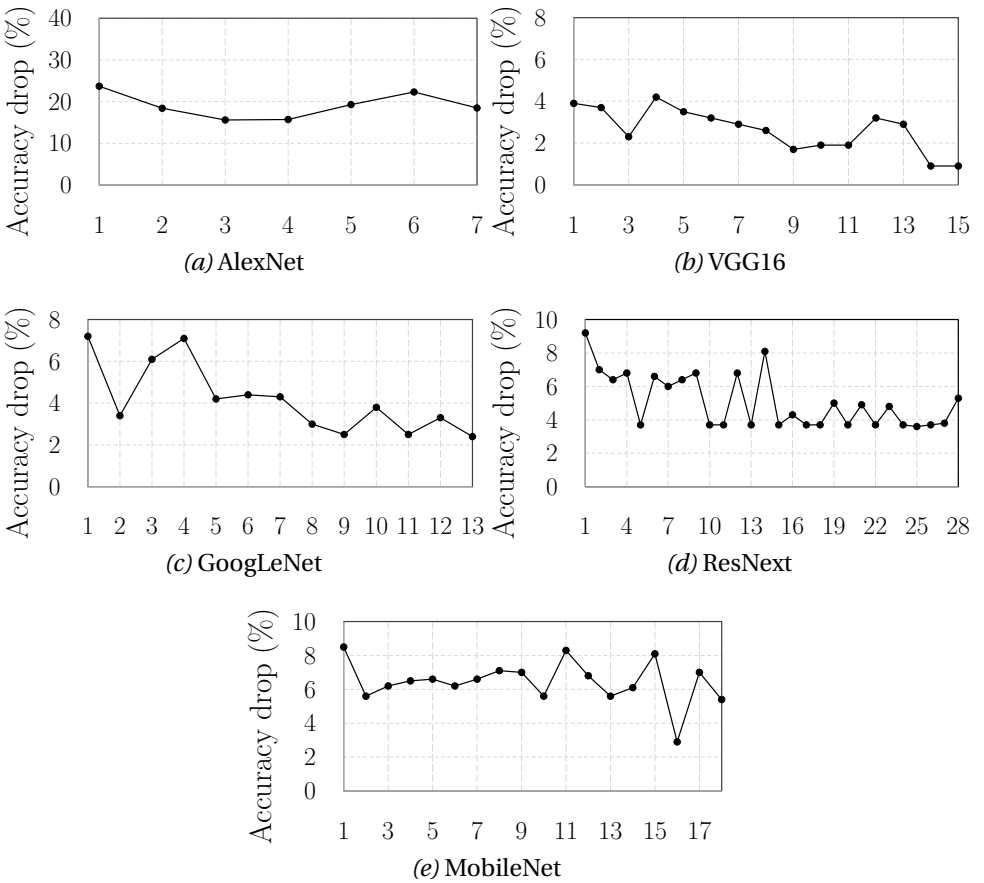


Figure 4.3: Sensitivity analysis performed on the five considered benchmarks. The accuracy drop obtained by instantiating, one at a time, a target approximate multiplier (i.e., MUL8VH in these plots) in each layer of the CNN models is measured and it is considered as a proxy of robustness.

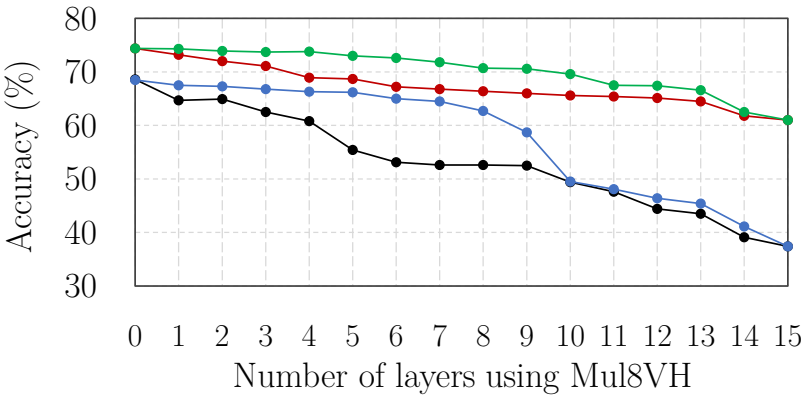


Figure 4.4: Achieved accuracy when using Mul8VH in an increasing number of layers of both VGG16 and an equivalent E^{2}CNNs implementation. Following the topological order of VGG16 layers (black) and E^{2}CNNs (red) to introduce approximation produces higher accuracy drops than introducing instead approximation following the order suggested by the proposed sensitivity analysis, both in the single-instance VGG16 (blue) and the corresponding E^{2}CNNs (green).

precious tool allowing a selection of candidate layers for the use of inexact arithmetic. Figure 4.4 confirms the benefits of this analysis by comparing the accuracy drop due to the use of Mul8VH in an increasing number of layers of the VGG16-CIFAR100 benchmark. The figure shows that simply introducing approximation in the VGG16 layers following a topological order (i.e., introducing the approximate multiplier from the first layer to the last one) performs much worse than introducing inexact arithmetic in an increasing number of layers following the order suggested by the sensitivity analysis. The major benefit of this second approach is that, for any given accuracy threshold, the number of layers executed using the approximate multiplier is higher, thus increasing the potential energy savings.

4.3.3 Analysis of the QoS/Energy trade-off

Table 4.1 shows that the two considered inexact multipliers consume 15% (MulF6B) and 50% (Mul8VH) less than the alternative exact implementation (Exact16), at the cost of introducing a relatively large arithmetic approxima-

tion. Clearly, more aggressive inexact multipliers like Mul8VH enable larger energy savings. Nevertheless, the use of Exact8, enabled by the heterogeneous quantization phase, offers energy savings of 85%, without introducing arithmetic errors in the computed products. The approximation in 4/8 layers is hence derived only from the quantization process that reduces the precision of weights and activations, while arithmetic operations do not generate any approximate results. Notice also that, at this point, the accuracy degradation due to this type of approximation is already evaluated during the quantization process. Therefore, the use of the Exact8 multiplier does not induce any further output quality reduction. These findings, combined with the more general observation that quantization has a lower impact on accuracy than inexact arithmetic, have driven the choice of pushing quantization as much as possible in the first stage of the proposed co-design optimization approach.

To appreciate the benefits of this methodology from an accuracy/energy perspective, I include in Figure 4.5 the results of an analysis conducted on the considered pool of benchmarks, showing the accuracy level, and the corresponding energy savings, achieved in each step of the presented strategy. In each plot, the black mark represents the baseline uniformly quantized single-instance CNN. These plots consider the Mul8VH as a target inexact multiplier. First, this model is transformed into an E^2CNNs equivalent (yellow mark). The main result of this transformation is an accuracy improvement ranging from 1.8% in AlexNet, to more than 6% in ResNext. Then, the results show that the largest fraction of energy savings is achieved by heterogeneously quantizing the E^2CNNs implementations, enabling Exact8 in all 4/8 quantized layers. The results reported in Figure 4.5 consider an accuracy threshold of 5% and show energy savings ranging from 22% in ResNext to more than 80% in VGG16.

Finally, the last stage of the proposed methodology adds inexact arithmetic in specific 8/16 quantized layers, by executing them using Mul8VH instead of Exact16 in the plots depicted in Figure 4.5. The accuracy-energy numbers obtained with our approach (green triangles) demonstrate that approximate operators can be effectively used to increase inference efficiency, allowing up to 20% more energy reductions. The amount of savings due to the intro-

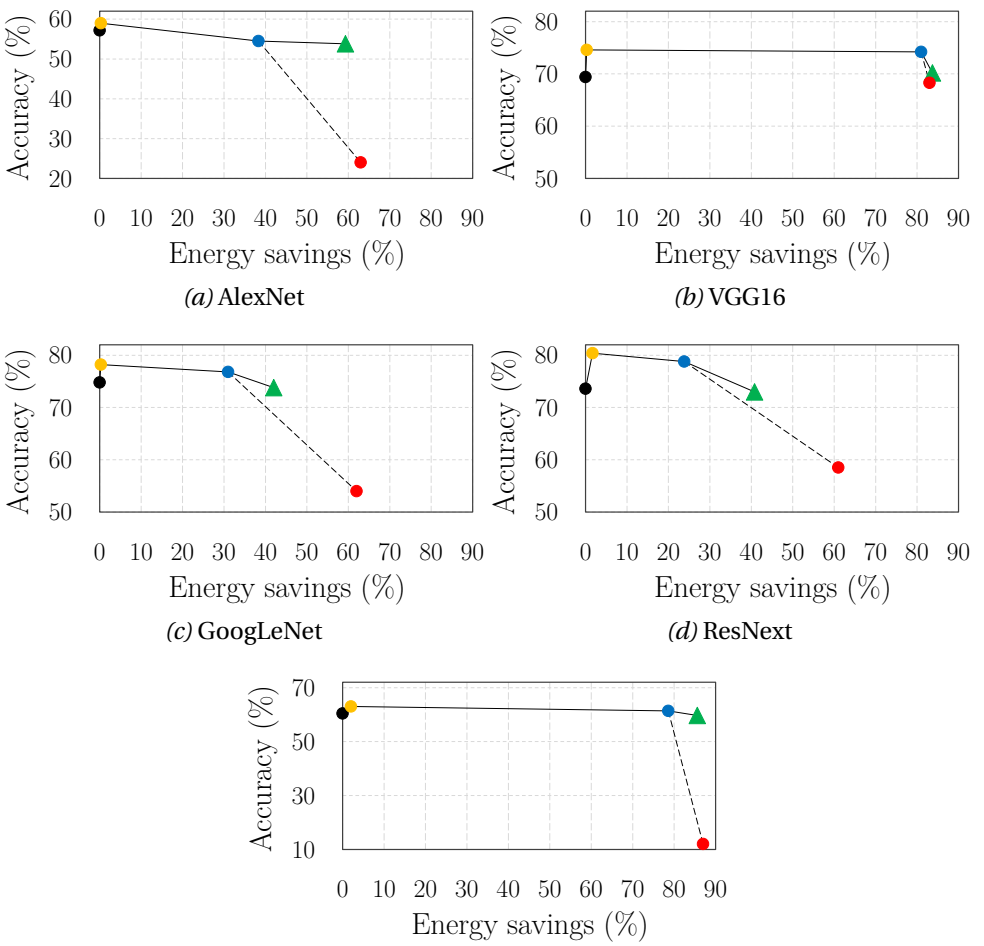


Figure 4.5: Accuracy/energy trade-off in the considered benchmarks considering Mul8VH as target inexact multiplier. The accuracy of baseline single-instance models (black) is improved by transforming them into E^{2}CNNs equivalents (yellow). A per-layer quantization strategy enables large energy gains by supporting the use of the Exact8 multiplier (blue). Further energy savings can be retrieved by the cautious use of approximate multipliers as suggested by the proposed solution (green), while an indiscriminate use of approximation in all 8/16 layers results in unacceptable accuracy drops (red).

duction of inexact arithmetic is linearly proportional to the number of 8/16 layers using the approximate multiplier. In turn, this number depends on multiple factors: on the one hand, it depends on the heterogeneous quantization applied in the first stage of this methodology. In fact, a high number of aggressively quantized layers and a large consumed accuracy margin return a lower possibility of mapping the remaining 8/16 layers into inexact arithmetic resources. This effect is clearly visible in VGG16, where most of the energy savings are due to quantization. On the other hand, the number of 8/16 layers that can be executed with arithmetic approximation depends on multiple factors: first, the lower the user-defined accuracy level, the higher the magnitude of approximation that can be injected. Second, the intrinsic degree of resiliency of the target architecture is directly proportional to the number of layers that can be approximated. Moreover, the higher the accuracy improvements of E^2CNNs, the higher the accuracy margin to trade-off by employing more aggressive quantization schemes and by using more frequently inexact arithmetic. When instead considering MulF6B as a target approximate multiplier, the proposed solution matches the fully inexact mapping strategy. This important result indicates that when considering multipliers that introduce a limited arithmetic error like MulF6B, the described methodology can maximize energy savings by allowing inexact arithmetic in all 8/16 layers. Nonetheless, it also demonstrates that this specific mapping strategy is not a viable solution in general, especially for tight accuracy constraints, as the conducted experiments show critical accuracy degradation in the case of an extensive use of Mul8VH.

Figure 4.5 indicates that heterogeneous quantization and approximate computing can be effectively combined to increase inference efficiency in CNN benchmarks. Yet, this combination should be carefully tailored. The green triangles representing the accuracy/energy characteristics of our optimized models show indeed that the two approaches can be effectively combined, by employing careful accuracy control in the optimization loop. Conversely, the approximation degree introduced when employing Mul8VH indiscriminately in *all* 8/16 layers can be too large, resulting in an unacceptable (and unpredictable) accuracy degradation (red circles). In some cases, this drop could be even higher when targeting more inexact multiplier implementations.

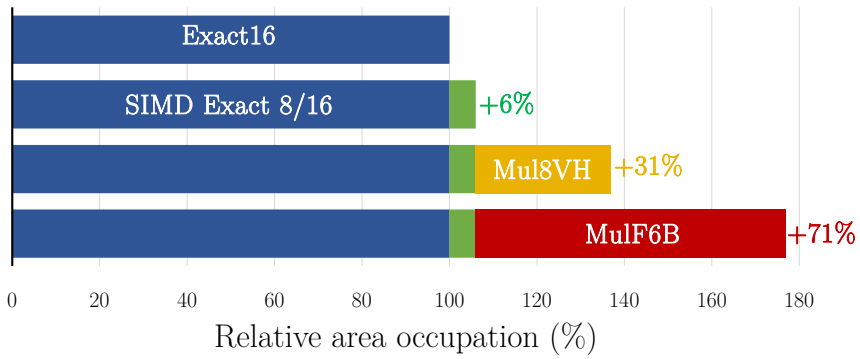


Figure 4.6: Area overhead relative to the Exact16 multiplier implementation. The overhead required to extend the Exact16 multiplier to enable SIMD executions of 8-bit multiplications is highlighted in green. The overhead for instantiating MulF6B and Mul8VH is illustrated in orange and red, respectively.

As a consequence, only an accuracy-driven optimization strategy can solve the challenge of properly mapping CNN layers into either exact or inexact arithmetic operators.

To conclude, the achieved results show 59.4%, 83.6%, 42.7%, 39.9%, 82.6% energy reductions for an accuracy degradation limited to 5% in AlexNet, VGG16, GoogLeNet, ResNext, and MobileNet, respectively, when compared to baseline exact single-instance implementations. Moreover, the proposed approach is able to retrieve up to 78% of the energy gains achievable when employing Mul8VH in all CNN layers, but preserving accuracy to user-defined levels.

4.3.4 Area overhead

A practical implementation of the proposed framework employing exact and inexact arithmetic requires the deployment of three multipliers: Exact8 is used for 4/8 quantized layers, while Exact16 and an approximate multiplier are used for 8/16 quantized ones. The Exact16 multiplier can be implemented by properly combining Exact8 units. In this way, 4/8 quantized layers can be executed using a Single-Instruction Multiple-Data (SIMD) approach, where the two individual Exact8 units operate independently on the input data (i.e.,

two 8-bit multiplicands and two 4-bit multipliers). Synthesis results of this SIMD multiplier design indicate that the additional logic and routing required to implement it incur just a 6% area overhead with respect to a non-SIMD 16-bit multiplier. In this way, the execution runtime of 4/8 quantized layers can be effectively halved.

The approximate multiplier must be instead instantiated as a second multiplier in this design. In general, the area overhead of this second multiplier can vary significantly, with highly inexact multipliers usually having a lower overhead than less approximate ones. Results considering the two evaluated approximate multipliers are summarized in Figure 4.6 and show that the area overhead for executing both exact and inexact arithmetic in the presented design can be limited to 31% (i.e., in the case of Mul8VH). Consequently, the ability of the proposed methodology to handle highly inexact multipliers can limit the area overhead. In fact, the deployment of Mul8VH in conjunction with the exact multiplier in the final design demands a smaller area footprint compared to less inexact (and thus larger) implementations such as MulF6B, which increases area requirements by 71%. At the same time, using Mul8VH still increases efficiency and guarantees a user-defined output quality thanks to a judicious per-layer mapping. Finally, although the trade-off between accuracy and efficiency could be explored more deeply by instantiating different approximate multipliers in different layers (i.e., according to their degree of resiliency), the obtained results indicate that the significant area overhead of these circuits may limit such an approach.

4.4 Conclusions

This chapter has discussed the effective use of approximate multipliers in quantized E^2CNNs designs. The proposed accuracy-driven methodology maps CNN layers into exact or inexact arithmetic resources to reduce energy while abiding by target output quality constraints. The illustrated approach can be extended to other deep learning models as well, such as Recurrent Neural Networks or transformers. More in general, any application that makes extensive use of multiply instructions can benefit from the proposed solution. In fact, being accuracy-driven, it can be applied to applications having different robustness levels, ultimately ensuring a desired QoS.

Still, the energy reductions at the system level can be very different depending on the actual physical implementation. In fact, the addition of an approximate multiplier in processor pipelines may not be advantageous. While being one of the most energy-hungry arithmetic units, the multiplier consumes far less energy than data transfers between the processor and the memory elements. As a consequence, despite the significant energy gains in the processor, the overall energy savings at the system level can be minimal. Conversely, this picture can be quite different when targeting hardware accelerators. For example, systolic arrays can be composed of hundreds of processing elements (PE), each necessarily instantiating a multiplier unit, among others. In these highly-parallel architectures, the energy cost is instead dominated by the arithmetic and logic operations of PEs, thus motivating the use of inexact arithmetic as a tool to reduce energy.

In-Memory Computing

5

In this chapter, the focus is on a novel class of hardware accelerators, that, due to their particular structure, mandate mindful co-design strategies. In-Memory Computing IMC represents a new computing paradigm that overcomes the usual Von Neumann computing structure by moving the execution of logic and arithmetic instructions very close to the memory elements. Doing so, IMC architectures become particularly appealing, as in-memory computation avoids energy-expensive data movements in-between processing and storage components. At the same time, the parallelism made available by the regular structure of memory banks presents a good opportunity to support the typical SIMD patterns of machine learning applications. Different IMC architectures have been proposed in the past years and a complete overview of the main proposed solutions is presented in the following section. My focus in this chapter is on a particular embodiment of IMC, named Bit-line Computing (BC). BC accelerators can be efficiently integrated into existing SRAMs. When two memory rows are accessed simultaneously, bit-wise operations (i.e., AND and NOR) between the bits of the two activated rows can be retrieved in the bit-lines. These logic operations between the two memory rows are then leveraged to derive more complex arithmetic operations, ultimately allowing the in-memory acceleration of compute-intense AI workloads.

In this regard, I would like to underline that my contribution to the studies presented in the next sections is mainly focused on the hardware-aware methodologies that optimize CNN models for their IMC inference execution.

Instead, the physical design and implementation of IMC resources have been proposed by Marco Rios, a colleague (and a *friend*) I have extensively collaborated with during the years of my Ph.D. As a result of a close collaboration between the two of us, as well as with other senior research members of the lab, a strong interaction between hardware design and application-level optimizations pervades the presented solutions. The description of architectural IMC solutions illustrated in the next sections is included anyway, as it is key to understanding my proposed approaches for improving the inference efficiency of the considered benchmarks.

5.1 IMC architectural design

5.1.1 Physical implementations

A wide variety of in-memory computing accelerators have been proposed in the last years, all aiming at blending computation and storage, but approaching the challenges of IMC system integration, technology, and analog vs. digital computation from different perspectives. Considering IMC integration in computing systems, IMC units have been proposed targeting different levels of the memory hierarchy, either interfacing the main memory [97] and/or caches [98], or as functional units augmenting processor pipelines and facing register files [99].

From a technology perspective, SRAM-based [98, 100] and DRAM-based [97] IMCs employ charge sharing, achieved by the concurrent activation of multiple memory rows. An alternative is represented by crossbar-based architectures, in which non-volatile programmable resistors are employed at the junction of horizontal and vertical wiring, modulating their connection. Both traditional memories and crossbars have been used to parallelize computations in the digital domain to realize logic gates [101, 102]. Nonetheless, they can also be employed as analog devices, where current, time, and voltage values represent inputs and outputs. In particular, Analog In-Memory Computing (AIMC) crossbars structures featuring non-volatile resistors are quite common accelerators in the field of AI, thanks to their higher density and energy efficiency [45]. AIMC crossbar architectures leverage arrays of digital-to-analog converters to encode inputs as voltage values that gener-

ate currents flowing through the crossbar. These currents then propagate through wires across resistances acting as programmable weights. Output currents are hence the result of analog matrix-vector multiplications, and are finally converted back into the digital domain by analog-to-digital converters at the crossbar periphery. Operations on AIMC cores are done in a fully parallel way, resulting in very potential speedups in workloads dominated by recurrent computing patterns based on dot products. Nonetheless, as computation is performed in the analog domain, outputs are affected by non-deterministic noise, deviating from expected results. This is due to a number of device and circuit non-idealities, including, but not limited to (a) device-to-device and cycle-to-cycle conductance variations, (b) output noise, (c) weight read noise, (d) IR drop, and (e) quantization noise [103, 104]. Moreover, these devices require the challenging co-integration of digital, analog, and non-volatile technologies.

A second strategy to implement IMC is that of Bit-line Computing (BC) [105]. This solution can be efficiently integrated into existing SRAM memories by adding the capability to concurrently activate multiple memory words. Bitwise operations can be then retrieved by employing sense amplifiers. As the name suggests, bit-line computing relies on the behavior of the bit-lines (BL and $\overline{\text{BL}}$) when two word-lines (WLs) are activated simultaneously, during the same cycle clock. Figure 5.1 illustrates this behavior, showing two SRAM bit-cells implemented as two cross-coupled inverters connected to the same BLs. Let's assume to activate simultaneously two WLs. If at least one of the two activated bit-cells in each BL stores the value '0', the voltage on the BL wire discharges to the ground through one (or both) access transistor(s) (M0) and the NMOS of their inverter. Only when both cells store the value '1' do the corresponding BLs remain at *Vdd*. Therefore, the BL connection behaves as the logic AND gate. Similarly, the negated BL signal ($\overline{\text{BL}}$) only retains *Vdd* if both cells store the value '0'. Therefore, $\overline{\text{Q}_0}$ and $\overline{\text{Q}_1}$ are both '1' in this case, ultimately implementing the functionality of a NOR gate.

This particular characteristic of bit-line computing accelerators can induce data corruption due to undesirable currents flowing among the memory cells. Previous studies have tackled this challenge in different ways: for example, the authors of [105] reduce the operating frequency, while the authors of [106]

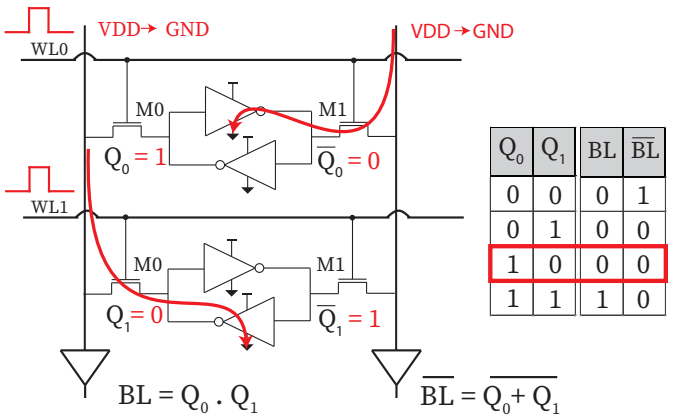


Figure 5.1: Bit-line Computing (BC) operating concept. Two word-lines are activated simultaneously and the discharge of the bit-lines results in the logical bit-wise AND and NOR between the accessed words.

propose 10T bit-cells as a more robust solution. Nevertheless, both solutions either decrease the performance or imply large area overheads.

My colleagues and I have recently shown [107, 108] that such operations can be reliably performed at high clock frequencies if the activated WLs belong to different sets of SRAM arrays, named Local Groups (LGs). This arrangement makes only memory cells in the same LG sharing short-distance vertical connections, named local bit-lines (LBLs). During read accesses, cell values are propagated via LBLs to the LG Periphery (LGP) circuit, which includes sense amplifiers and read/write ports. In-memory bit-line operations are then retrieved at the outputs of these sense amplifiers, thus protecting memory cells from data corruption. A more detailed discussion of this approach is presented in the next sections. The presence of bit-wise AND and NOR logic operations at the output of the BL sense amplifiers can be leveraged to implement arithmetic functionalities. For example, additions can be derived at the array periphery with limited additional logic. Multiplications can be also implemented as a sequence of shift-add operations between two memory words, ultimately allowing in-memory computing to execute a wide range of applications at the cost of minimal overhead at the memory periphery.

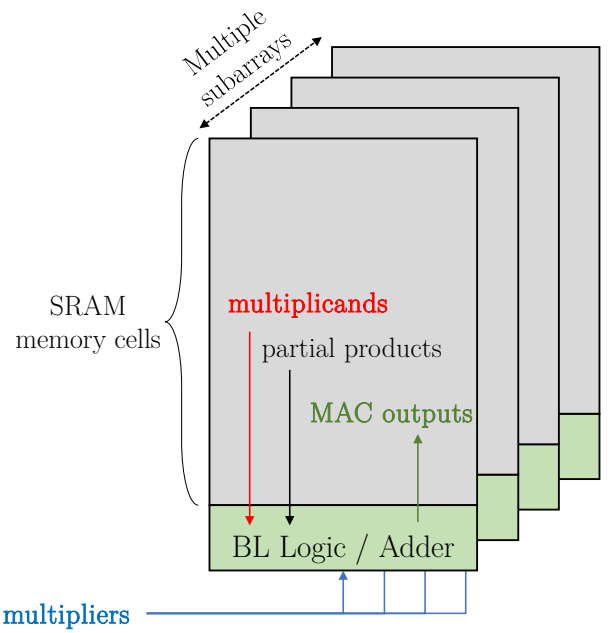


Figure 5.2: IMC architecture comprising multiple subarrays. Multiplications are derived by leveraging the adder and shifter units included in the memory periphery (green area). Multiplicands and the computed partial products reside inside the IMC, while multipliers are streamed bit-by-bit to compute the partial products.

5.1.2 Implementing multiply-accumulate operations

In the previous section, I briefly discussed how arithmetic instructions such as additions and multiplications can be effectively implemented at the periphery of memory arrays, using the bit-wise AND and NOR as baseline signals. Herein, I present a more detailed description of the implementation of multiply-accumulate (MAC) operations. Then, in Section 5.2, I introduce a co-design methodology targeting a novel BC architectural solution, which, from a HW-SW co-design perspective, I then exploit to enable an efficient execution of optimized AI workloads. My colleagues and I have presented this work in [101].

An overview of the proposed IMC architecture implementing MAC instructions is illustrated in Figure 5.2. It is structured as an SRAM memory array

composed of multiple subarrays, each comprising Local Groups (LGs) as in [98]. In this way, multiple operations among operands in the same subarray, but different LGs, can be performed in parallel. The figure also shows the arrangement of multiplicands and multipliers (i.e., input operands of multiplications): multiplicands are stored inside the memory elements and are hence defined as In-Memory Operands (IMOs). Conversely, multipliers are broadcasted bit-by-bit inside the IMC periphery (Broadcasted Operands, BOs). Indeed, as previously presented in Section 5.1, multiplications are implemented as a series of shift-add instructions. In this way, one bit of the multiplier generates a partial product, that has to be repeatedly accumulated to compute the final output. This arrangement enables the parallel execution of different multiplications in multiple IMC subarrays, by broadcasting the same bit of a multiplier to different subarrays (i.e., to multiply in parallel the same BO with different IMOs). Importantly, the bit-by-bit streaming of multipliers allows them to employ arbitrary bitwidths, thus supporting very heterogeneous quantization schemes.

IMOs are latched when first read to improve performance so that repetitive accesses to the same memory words (and the associated energy cost) are avoided when performing multiplications. In fact, by using latches (rather than flip-flops) the fetching of an in-memory operand and the first shift-add operation can be performed in the same clock cycle. Since results of in-memory operations are stored in registers inside the IMC periphery (as opposed to SRAM cells), write-backs to the SRAM memory array are avoided during the calculation of products. 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. The result is a 2× acceleration of multiply operation compared to state-of-the-art bit-line computing architectures [98].

At the circuit level, local groups in subarrays have their own set of local bit-lines (LBL and LBLb). Then, depending on the actual operation, read/write ports connect LBLs connect to Global Bit-Lines (GBL). To read a word or perform an IMC operation, both LBLs and GBLs are pre-charged. Consequently, a voltage-based sense amplifier inside the LG asserts a logic value controlling whether the GBL is discharged (or not) once the word-line is activated. Write

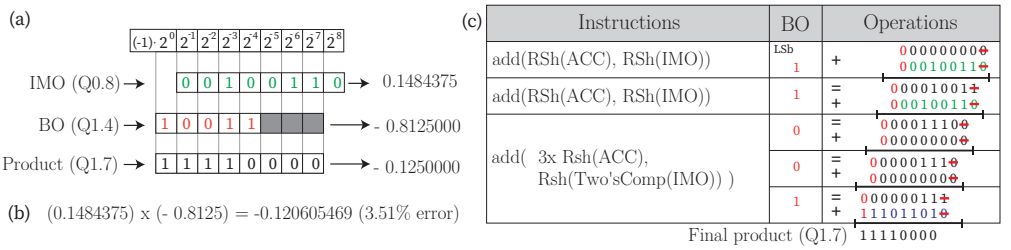


Figure 5.3: Multiplication between IMO in $Q0.8$ and BO in $Q1.4$ fixed-point formats, with the product represented $Q1.7$ (a). The performed multiplication produces a 3.51% error (b), because it is implemented as an iterative approximate algorithm based on shift-add operations (c).

operations are performed by setting up the data on the GBLs and activating the WL and the write port at the LG Periphery (LGP).

5.1.3 Multiplications as a series of shift-adds

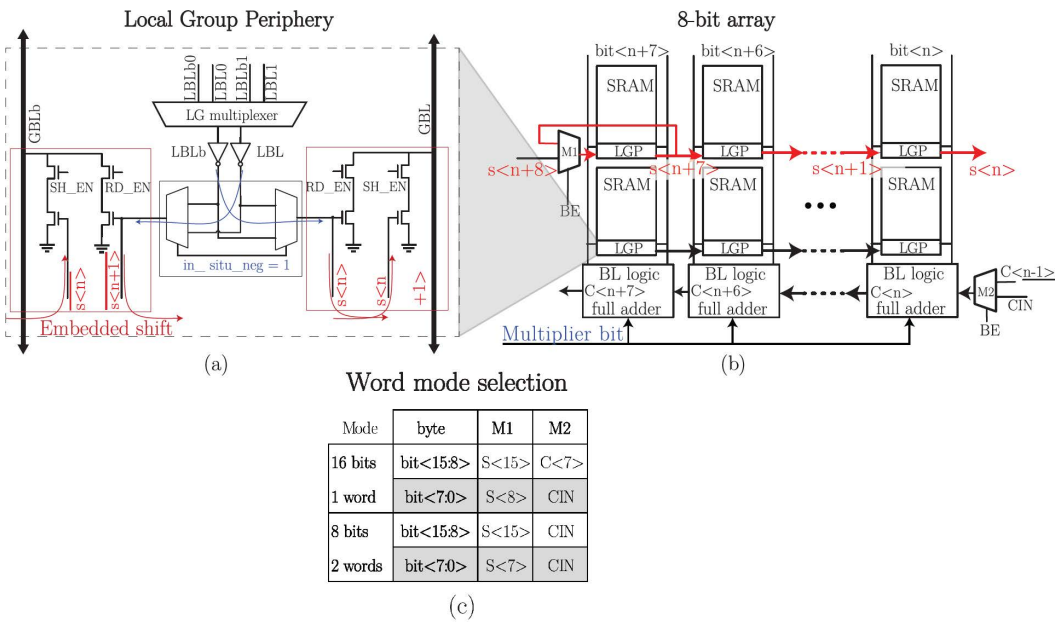
The IMC architecture designed by Marco Rios, and herein used as a target hardware resource for the proposed CNN optimizations, supports MACs in the form $(\sum_{i,j} a_i \times b_j)$, in fixed-point format and between values in the range $[-1,1)$ encoded in two's complement. We presented embodiments of such an architecture in [101, 109].

Fixed-point values are usually defined according to the $QN.M$ notation, where N represents the number of integer bits (including the sign bit), while M represents the number of fractional bits. For example, in [109], we scale weight values in the range $[-1,1)$, so that they can be represented using a $Q1.M$ representation. Instead, input activations are assumed to be unsigned real numbers spanning the range $[0,1)$, thus encoded in the $Q0.M+1$ format. Figure 5.3 provides a step-by-step illustration of the operations performed to execute multiplications. The multiplicand and the output result have the same bitwidth since the latter is then stored inside the IMC arrays. Moreover, the example shows that the right-shifted bit (i.e., the least significant bit, LSb) is not always zero, thus leading to approximate products.

Still, the implemented multiplication is safe by construction from overflows, even if it potentially induces rounding errors. Indeed, the multiplication depicted in this example produces a 3.51% error in the output product. Nonetheless, I will show in Section 5.2.1 that this approximation leads to negligible accuracy drops in the evaluated workloads. Notice that, the instruction involving the Most Significant bit (MSb) of the multiplier adds the right-shifted partial product to the *two's complement* of the IMO (Figure 5.3-c). Compared to an unsigned multiplication, the implementation of the two's complement multiplication requires extra operations, as each partial product is computed using two right shifts and one addition. Moreover, two new operations have to be implemented in the IMC periphery to effectively support this algorithm: in-situ right-shift and in-situ negation (to compute two's complements). To speed up the execution time of multiplications and, as a result, the execution time of MACs, we have shown in [107] how the right-shift and negation instructions can be executed concurrently with the addition.

This challenge has been addressed by the architectural solution illustrated in Figure 5.4. First, an embedded shift circuit is introduced in the LGP to enable the right bit shift in the register storing the partial products (Figure 5.4-a). Then, the read ports inside the LGP are extended, so that the output from the adjacent sense amplifier can control the discharge of the GBL (red arrows). Figure 5.3-c represents the right-shifted bit with a red hyphen. As for all the bits in the register, the MSb is right-shifted as well, but is also kept in the MSb position when the shifted value is represented in two's complement.

The two's complement of multiplicands is computed by negating their value inside the LGs. To implement this feature, two multiplexers that can invert the LBL and LBLb signals are employed, as illustrated by the blue arrows in Figure 5.4-a. The carry-in of the adder in the subarray periphery must be set to '1', and by chaining in-situ two's complements and shifts to additions, extra clock cycles are avoided. As a consequence, all in-memory operations take two clock periods: the first one is used to read the operands and perform the in-memory operation computation, while the second one is used to write back the result inside the memory subarray.



Word mode selection

Mode	byte	M1	M2
16 bits	bit<15:8>	S<15>	C<7>
1 word	bit<7:0>	S<8>	CIN
8 bits	bit<15:8>	S<15>	CIN
2 words	bit<7:0>	S<7>	CIN

(c)

Figure 5.4: (a) The Local Group Periphery (LGP) schematic: blue arrows represent the data path for the in-situ negation operation, while red arrows show the functionality of the in-situ right-shift. (b) architectural overview of an 8-bit array in the proposed IMC design. (c) Multiplexers M1 and M2 output for the two-word modes: either one 16-bit word or two 8-bit words.

Word-level parallelism represents a second feature supported by the designed IMC architecture. In fact, it is possible to consider the stored values as either a single 16-bit word or two 8-bit words in the memory location of each subarray. When representing multiplicands (stored inside the IMC array) using just 8-bit operands, the corresponding workload is effectively halved, because two multiplications are performed in parallel in each subarray. On one side, read and write operations are executed in the same way, regardless of the selected mode (either two 8-bit IMOs or one 16-bit IMO). On the other hand, the in-memory right-shift and addition must be properly adjusted to correctly implement word-level parallelism. Figure 5.4-b presents the circuit that enables the selection of one of these two configurations, exemplifying it with one 8-bit (bit<n+7:n>) word. To implement a single 16-bit operation, two of the illustrated 8-bit blocks are assembled. First, the mode selection is driven by two multiplexers in each 8-bit block. Considering a subarray that stores 16-bit words (bit<15:0>), the table depicted in Figure 5.4-c shows the outputs of the two multiplexers (i.e., M1 and M2) for each of the 16-bit arrays. The M1 multiplexer controls the shift-right input on the MSb, and, as previously discussed, the bit shifted in the bit<15> is always the MSb (i.e., s<15>). Conversely, the shifted bit in the bit<7> is either s<7> for 8-bit modes, or s<8> for 16-bit modes. The carry-in on the LSb (i.e., CIN in the figure) is '0' for typical additions and '1' for the addition with the two's complement of the IMO. Therefore, the carry-in of the bit<8> is either the carry C<7> for the 16-bit mode, or CIN for the 8-bit one.

5.1.4 Accelerating CNN layers

In the previous section, I presented a circuit included in the IMC array periphery to implement two's complement multiplications, considering IMOs as either a single 16-bit word or two 8-bit ones. Therefore, it can also support efficient execution of MAC operations between in-memory operands acting as multiplicands and broadcast operands, acting as multipliers. The computed MAC outputs are then stored back in the memory array. The execution of matrix-vector operations requires minimal HW extension and is extremely easy to implement. First, the input operands of dot products are mapped as IMOs or BOs. Then, IMOs are initially loaded into the memory arrays, and finally, BOs are streamed inside the LGP to perform MAC oper-

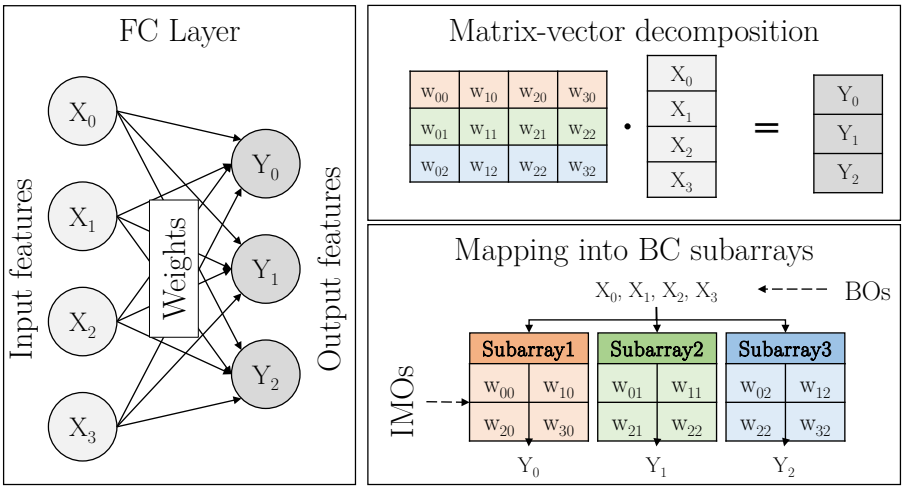


Figure 5.5: Overview of a fully-connected layer, with output features computed by applying a weight matrix to the input feature vector (left). This layer is naturally represented as a matrix-vector operation, whose execution can be parallelized in the multi-subarray structure of the proposed IMC architecture (right).

ations. In line with what I have presented in previous chapters, I focus on deep learning workloads. In particular, considering Convolutional Neural Networks (CNNs) models, the in-memory acceleration of convolutional and fully-connected (FC) layers is a natural target for the inference optimization of these benchmarks. In fact, both types of layers can be implemented as matrix-vector operations and represent the computing bottleneck of these models. Therefore, their acceleration in IMC devices offers striking inference speed-ups.

Fully-connected layers multiply an input feature vector with a weight matrix, to compute the output feature vector, as illustrated in Figure 5.5(left). The figure also shows how these layers can be naturally decomposed into matrix-vector operations (top-right). To effectively accelerate these layers into the proposed IMC arrays, the different IMC embodiments we have recently proposed in [101, 109, 110] to map FC layers in multiple subarrays, as shown in Figure 5.5(bottom-right). In this way, the same input features are broadcasted to multiple subarrays simultaneously, to allow a parallel execution of dot products. Weight values are stored inside the IMC arrays to act as

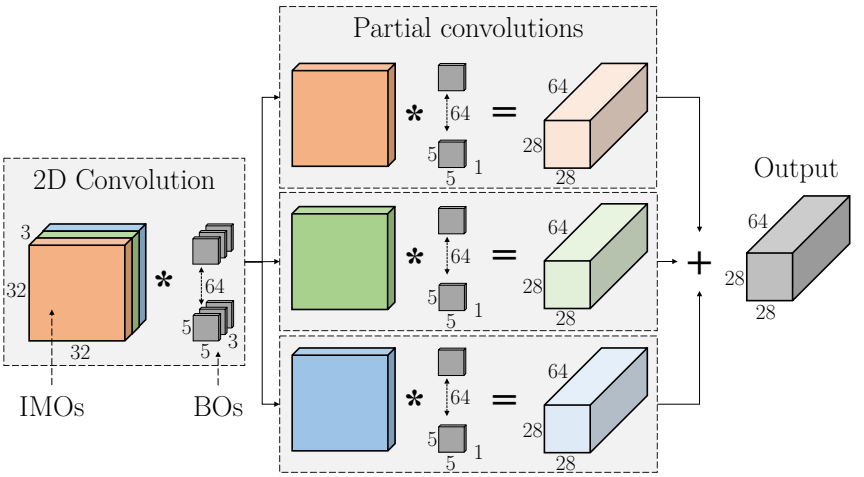


Figure 5.6: Decomposing convolutions into multiple partial convolutions (mapped into multiple subarrays) to speed up their execution and fit into individual subarrays.

IMOs, while input features are streamed as BOs. Mapping weights as BOs and activations as IMOs maximizes data reuse of weights, ultimately speeding up matrix-vector operations.

Conversely, 2D convolutional layers (i.e., the ones typically used when processing images) apply a 4D-filter tensor to 3D input features. Each filter consists of a 3D weights matrix having the same depth as the input feature, but a much lower width and height, which determines the 2D kernel size (e.g., 3x3 or 5x5 are common kernel sizes in modern CNNs). The convolution of a filter with an iso-dimensional fraction of the input feature map produces one single output element. Then, the filter is slid over the input feature matrix to compute all output elements of the corresponding output channel. As a result of this computing pattern, each filter generates a one-channel feature matrix, so that the number of filters determines the number of channels at the output of the layer. Due to the large size of convolutional layers in most CNN architectures, they are decomposed into partial convolutions to be effectively run in the target IMC architecture. The approach my colleagues and I have proposed to map these layers in the presented BC architecture is summarized in Figure 5.6. To accommodate the size of the subarray, individual input channels are stored in different subarrays. Weights of convolutional filters are

Table 5.1: Summary of weights and activations mapping as In-Memory Operands (IMOs) and Broadcasted Operands (BOs) in convolutional and fully-connected layers.

Operands	**Layer Type**	
	Convolutional	Fully-Connected
Multiplicands (IMOs)	Activations	Weights
Multipliers (BOs)	Weights	Activations

then broadcasted as BOs to execute dot products. Finally, the partial outputs must be summed up together to construct the output of the layer. The computing pattern of convolutions is not directly expressible as a matrix-vector operation. Nevertheless, previous works have demonstrated that an algorithmic and data transformation named *im2col* [28] can be applied to represent these layers as matrix-vector operations. As multiple filters are applied on the same input feature map, it is natural to map weights as BOs, to apply them in parallel to input features stored in different subarrays (IMOs) as shown in Figure 5.6. An opposite mapping is used for FC layers, to improve data reuse and thus performance. Indeed, weights and activations act as IMOs and BOs, according to the type of layer. A summary of this mapping strategy is shown in Table 5.1.

So far, I have initially shown how bit-wise logic operations can be retrieved at the output of BL sense amplifiers and how these outputs can be combined with low-overhead logic to implement multiplications at the periphery of memory elements. Then, I have described how the most compute-intense layers of CNN inference workloads can be mapped and executed in the presented IMC architecture to speed up inference runtime. Although arithmetic multiplications require several cycles when implemented as a series of shift-add operations, high performance can be achieved. On the one hand, word-level parallelism can be leveraged in each subarray by operating IMOs as two 8-bit words instead of a single 16-bit one. On the other hand, partitioning the SRAM into different subarrays enables the parallel execution of MACs between IMOs of different subarrays and the streamed BO.

Additionally, workload optimizations can (and should) be also employed to reduce the bitwidth of streamed operands, ultimately reducing the number

of cycles required to execute multiplications. Indeed, the effectiveness of word-level parallelism and workload optimization is highly influenced by algorithmic-level optimizations of CNN models, such as quantization and pruning. In fact, both methods can effectively reduce the number of shift-adds, either by reducing the bitwidth of BOs (i.e., quantization) or by reducing the number of multiply instructions (i.e., pruning). Nevertheless, current works on quantization do not explore this interdependence in detail. As an example, the authors of [54] adopt a fixed 16-bit representation for activations and an 8-bit one for weights. This approach does not suit well the proposed IMC architecture, where an opposite quantization scheme would be more advantageous in FC layers.

Therefore, I have proposed a novel heterogeneous quantization scheme for the in-memory execution of CNN inferences in [101]. This methodology, presented in the next section, underlines the importance of a HW-SW co-optimization, where a hardware-aware model optimization can be effectively leveraged in the underlying resources. I will show that important efficiency gains can be harnessed when employing aggressive stances such as per-layer quantization schemes, and that these can be effectively supported with little hardware overhead.

5.2 An IMC-aware CNN quantization methodology

Quantizing edge AI benchmarks is beneficial from a storage perspective, as it alleviates memory constraints, as well as from a runtime perspective, as it also reduces the computing complexity of these models. This second perspective is particularly true when employing HW units that can efficiently exploit low-bitwidth operands to improve both energy efficiency and inference runtime acceleration. Common quantization schemes employ uniform quantization on 8 or 16 bits, as these schemes are extremely effective for accelerators supporting integer arithmetic. Nevertheless, the peculiar architecture of the proposed In-Memory Computing (IMC) accelerator enables even larger gains when more fine-grained quantization schemes are used. In fact, data parallelism can be achieved by quantizing In-Memory Operands (IMOs) to just 8 bits, and the execution time of individual MAC operations can be reduced by more compact quantization levels in the Broadcasted

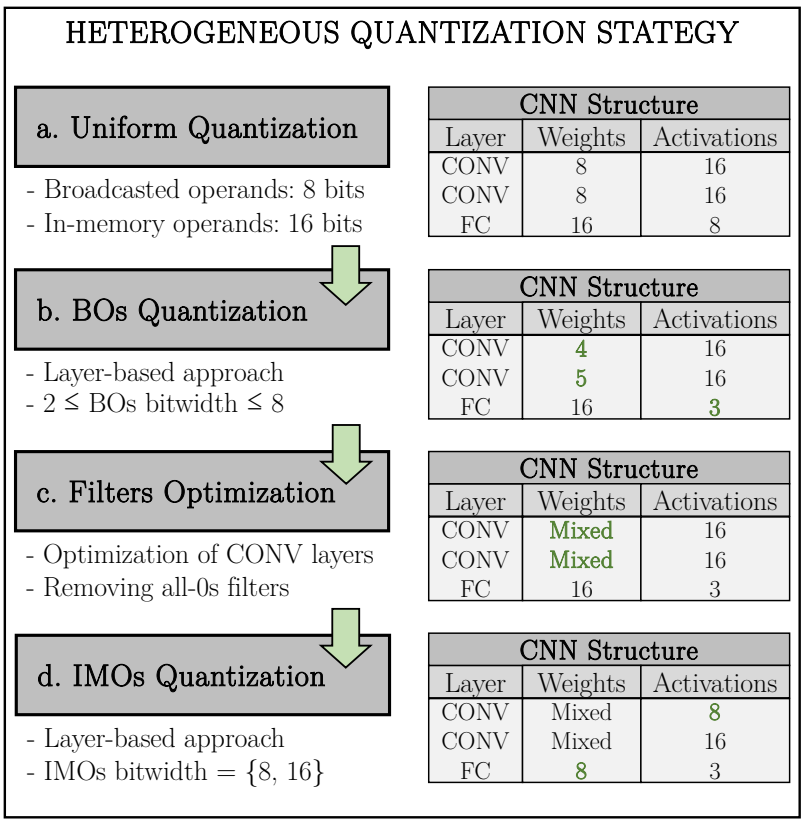


Figure 5.7: Hardware-aware quantization methodology (left). Running example showing the target operands optimized during each phase (right).

Operands (BOs). With this in mind, I have proposed in [101] the quantization methodology summarized in Figure 5.7. For simplicity, the example shown in figure illustrates a CNN model featuring only two convolutional layers and a fully-connected one.

The bitwidth of both multiplicands and multipliers is adjusted in a multi-stage process that leverages the flexibility of the presented IMC architecture to optimize runtime and efficiency. Input of this quantization strategy is a uniformly quantized CNN model, having 16-bit multiplicands (i.e., IMOs) and 8-bit multipliers (i.e., BOs), as shown in Figure 5.7-a. As already discussed, this quantization scheme demonstrated to achieve iso-accuracy levels when compared to floating-point, as also shown in [106] and [54]. I consider these

homogeneously quantized models as baselines to evaluate and compare the resulting CNN models optimized with the proposed strategy.

The first step of the proposed quantization strategy tailors the bit-width of BOs in each layer independently (Figure 5.7-b). To this end, the bitwidth of BOs is iteratively reduced starting from the layer having the highest number of MAC operations (and therefore the higher potential for savings). To compensate for the approximation induced by the bitwidth reduction of the BOs in a certain layer, the network is then re-trained for a small number of epochs. Then, the accuracy of the resulting model is checked: if the accuracy degradation exceeds a user-defined threshold, the previous BOs bitwidth is re-employed. In the experiments presented in Section 5.2.1, I consider a threshold of 1% to show how significant energy savings can be achieved without sacrificing accuracy. The optimization loop proceeds by considering all the convolutional and fully-connected layers of the CNN, trying to reduce the bitwidth of BOs by one bit at each optimization step. Once all layers have been processed, the described iteration repeats, trying to further reduce the BOs bitwidth in all layers from which the methodology hasn't previously backtracked (i.e. because of unacceptable accuracy values). When no further bitwidth reduction is possible, this first phase ends.

Focusing on convolutional layers, it turned out that a large number of filters do not use the entire value range for weights representation, especially after the above-mentioned heterogeneous quantization procedure. As illustrated in Figure 5.7-c, the proposed method drops, without loss of accuracy, the most significant bits in each filter if allowed by weight ranges. For example, if the value range of a filter is $R \subset [-0.25, 0.25)$, two MSbs can be dropped, and outputs should be divided by $2^{Dropped\_bits} = 4$. It also happened to obtain filters having all weights equal to zero. Thus, this stage removes them from the CNN model, without incurring any accuracy degradation.

Finally, the last step of the optimization flow (Figure 5.7-d) reduces the bitwidth of IMOs. In this way, it leverages the word-level parallelism supported by the memory arrays of the IMC architecture. Similarly to the approach adopted for adjusting the bitwidth of BOs, the bitwidth of IMOs is tailored on a per-layer basis. However, in this case, only two bitwidths are

admitted (i.e., either the baseline 16-bit IMOs or more compact 8-bit IMOs), which correspond to either one or two values per memory word, respectively. While, in principle, this approach could be extended to four 4-bit values or eight 2-bit values per word, such settings would incur unacceptably large accuracy degradations and higher area overheads.

5.2.1 Methodology evaluation

5.2.1.1 Experimental Setup

An IMC architecture composed of 32 subarrays of 640 bytes each and interfaced to an external read/write port using an H-tree interconnect is considered in the proposed experiments. With a methodology similar to the one presented in [98], the proposed IMC architecture is implemented as a full-custom design, evaluated using hspice energy and timing characterization. Implemented using a 28nm CMOS technology from TSMC, the architecture can operate at a maximum frequency of 2.2 GHz. In each subarray, read and write operations of 16-bit words cost 376pJ and 414pJ, respectively, while an in-memory shift-add instruction consumes 381pJ. Each subarray has an area of $1240\mu m^2$, of which 26.5% is used for the in-memory computing logic and the local group periphery circuit. Only 6.4% of the total area is used to implement the in-situ negation, embedded shift, and word-level parallelism, while 67% of the area is composed of SRAM cells.

A cycle-accurate simulator[1] introduced in [101] is employed to emulate the execution of convolutional and fully-connected layers from runtime and energy perspectives. To implement the mapping strategy discussed in Section 5.1.4, the simulator tiles the input of each layer and distributes the tiles to different subarrays, employing multiple rounds if the number of tiles exceeds the number of subarrays. A data transfer bandwidth of one word per cycle for writing inputs and reading back results is assumed. Conversely, IMC operations require two cycles: the first one is to perform the IMC operation, while the second one is to write back the result. These are executed in parallel on each subarray.

[1] https://www.epfl.ch/labs/esl/research/open-source-tools/cnn2blade/

The CNN optimization methodology is evaluated from an accuracy perspective considering the following CNN benchmarks: LeNet5 [70] is evaluated on CIFAR-10 [73], while AlexNet [72], VGG16 [83], MobileNet [76], and Xception [111] are instead evaluated on the more challenging CIFAR-100 dataset [73]. CNNs are first trained in PyTorch [85] using floating-point precision for 200 epochs, obtaining accuracies in line with the state-of-the-art. Similarly to [98], models are then homogeneously quantized to 16-bit multiplicands and 8-bit multipliers and refined for 20 additional training epochs using the quantization functions described in [77], ultimately resulting in no accuracy loss. To establish a baseline, the same re-training procedure is repeated to further homogeneously reduce the bitwidth of multipliers (i.e., BOs) from eight bits down to two bits, re-training the models for five fine-tuning epochs at each step. Five re-training epochs are also run after each optimization step in the proposed methodology following phases (Figure 5.7-b and Figure 5.7-d).

The presented accuracy numbers, as well as all the accuracy results presented in the following sections, are evaluated via the C++ inference solver described in Section 2.3.1, extending its features to better emulate inference processes as executed in the proposed IMC architecture. Indeed, in addition to its ability to emulate real fixed-point arithmetic, I have substituted fixed-point multiplications with a new implementation based on the iterative use of shift-add operations, as performed in the compute arrays and illustrated in Figure 5.3.

5.2.1.2 Experimental Results

First, I have conducted experiments to evaluate the impact of the implemented multiplication algorithm (see Figure 5.3) on the accuracy of CNN models. Indeed, the in-memory implementation of multiply instructions is based on an algorithm that iteratively truncates partial products. I have measured the average magnitude of errors introduced by this approximate implementation of multiplications by executing them, over all possible input combinations of IMOs and BOs. The obtained results indicate that it introduces a Mean Relative Error (MRE) of only 1.6% in the output results, so that the effect on output products of such implementation is negligible, on

average. Most importantly, it also causes an insignificant average accuracy degradation of just 0.11% in our evaluated benchmarks.

The trade-off between inference speed-ups, illustrated as a reduction in the number of required clock cycles, and the corresponding accuracy degradations of different quantized solutions, is summarized in Figure 5.8. Energy trends closely follow performance ones. Indeed, inference runtime is reduced as a result of lower operand bitwidths, which, in turn, lowers both the amount of data transfers and the number of MAC operations. Black lines report the achieved accuracy of baseline CNNs having a uniform BOs bitwidth, ranging from eight (leftmost points) to just two bits (rightmost points). For example, a BO bitwidth of five bits results in energy savings of 35%, with an average accuracy degradation across the evaluated benchmarks of 1.3%, which rapidly increases for further bitwidths reductions. Conversely, green lines report the cycle-count reduction and the corresponding achieved accuracy of CNN models optimized at different steps of the proposed methodology, considering an imposed accuracy drop threshold of 1%.

Points (b) and (c) in Figure 5.8 show that a heterogeneous quantization of BOs and the consequent filter-level optimization of bitwidths in convolutional layers already improve significantly the accuracy/energy trade-off in most benchmarks. Indeed, the optimized models incur very low accuracy degradations, achieving at the same time efficiencies comparable to (or higher than) those obtained with a homogeneous 2-bit quantization. Such a favorable performance/accuracy trade-off is the result of the flexibility enabled by the proposed hardware, which effectively supports the computation of highly heterogeneous CNN models. In particular, the filter-level optimization, represented as step (c), allows for large BOs bitwidth reductions with no effects on accuracy. For example, 27% and more than 70% of filters are removed in LeNet5 and VGG16, respectively, as they contain only *0*-valued weights. The obtained results also indicate that the likelihood of finding prunable filters is higher in larger models like MobileNet and VGG16 than in smaller models like LeNet5. More importantly, such a ratio is dramatically increased by aggressive quantization, as small values are cast to *0*. Indeed, just 3% of filters contain only *0*-valued weights in floating-point models, on average.

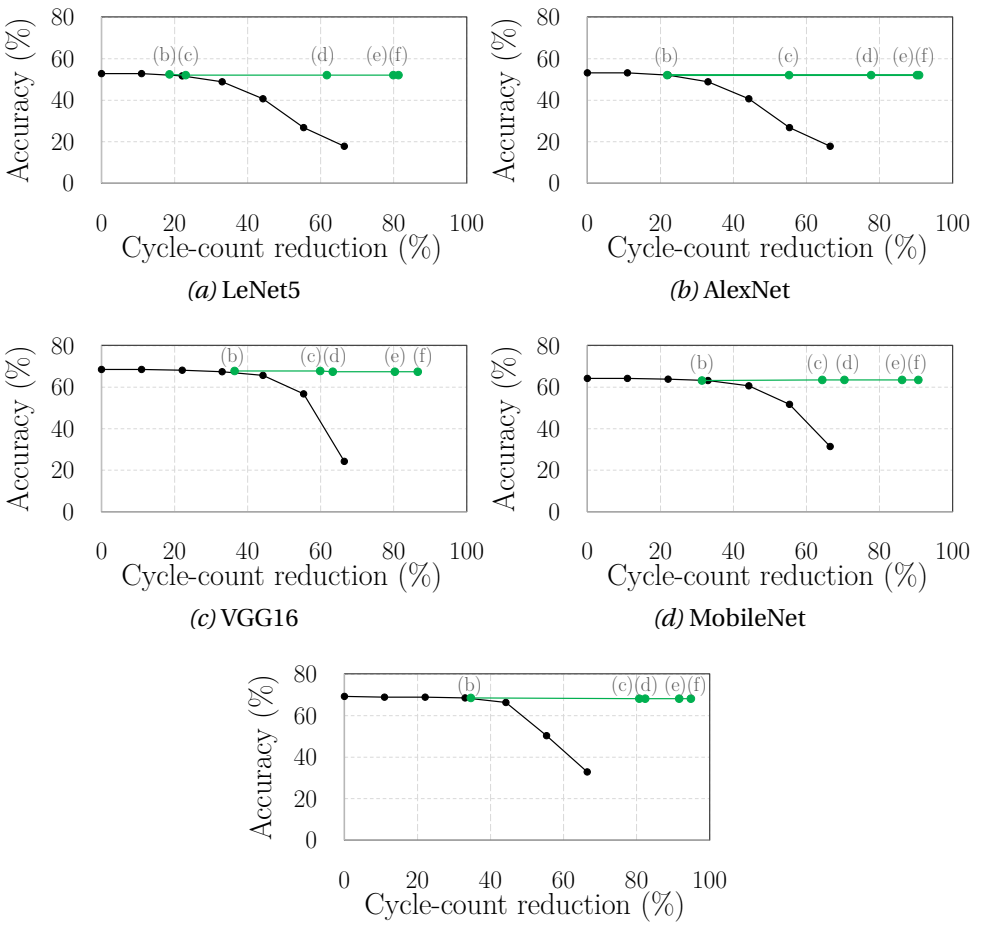


Figure 5.8: Achieved cycle-count reductions and corresponding accuracy level in uniformly quantized benchmarks (black) and in CNNs optimized using the proposed methodology (green). Letters (b-d) correspond to the optimization stages of the proposed strategy, as indicated in Figure 5.7. Letters (e, f) refer instead to HW-level optimizations including multi-bit shifts performed by the Embedded Shift (e) and the skip of MACs involving *0*-valued BOs (f).

Points (d) illustrate the additional savings that can be achieved by reducing 16-bit IMOs to 8-bit IMOs in certain layers. Gains are variable and depend on multiple factors, such as the size of the optimized layers and the accuracy margin. Finally, points (e) and (f) are not a direct consequence of the presented heterogeneous quantization methodology, but instead the result of features offered by the proposed IMC design. In particular, step (e) represents the benefits of the embedded shift, which allows up to three single-cycle bit-shifts, as shown in the multiplication example shown in Figure 5.3. Additionally, in (f), MAC operations involving zero-valued broadcasted operands are skipped.

Overall, the obtained results show that the number of cycles to perform convolutional and fully-connected layers in the proposed IMC design can be reduced to more than 90% when optimizing benchmarks according to the presented quantization methodology. Moreover, these results can be obtained by retaining accuracy within 1% of the baseline. The obtained results demonstrate the central importance of co-integrating SW optimizations with HW design. In fact, the designed IMC architecture efficiently supports the use of custom BO bitwidths and, dually, the heterogeneous quantization strategy leverages the hardware functionalities to tailor the bitwidth of BOs and IMOs accordingly.

5.3 Managing overflows in IMC

In the previous section, I presented an In-Memory Computing (IMC) architecture based on the concept of bit-line computing (BC), that can execute additions and multiplications in the periphery of SRAM arrays. I have described how its functionalities have been extended to efficiently support multiply-accumulate (MAC) operations, dot products, and, ultimately, matrix-vector operations. An interesting aspect of the multiplication algorithm based on shift-add operations is that it does not generate overflows by construction, but only results in approximations having a negligible impact on accuracy. Nonetheless, when calculating dot products, overflows can appear during the accumulation of computed products, thus affecting the reliability and robustness of a target application. In this section, I discuss this aspect, usually under-looked in previous works. I will show how the previously proposed

IMC architecture can be easily extended to prevent overflows in dot product computation, showing how well this improved implementation compares to other baselines. I will also present a more general workflow for the in-memory acceleration of CNN inference, combining operand scaling functions with an improved quantization strategy to better adapt to the new hardware architecture. This analysis has been recently presented in [109].

5.3.1 Numerical overflows

In computer science, overflows are familiar occurrences that can generate severe issues in numerical calculations. An overflow happens when a calculation (e.g., addition or multiplication) produces a result that is too large to be represented by the data type used to store it. This is more common when dealing with integer or fixed-point representations, but it may also affect floating-point arithmetic.

In integer representation, overflows occur when the result of an arithmetic operation exceeds the maximum value that can be stored using a certain data type. For example, let's assume the addition of two 16-bit signed integers. The maximum value they can represent is composed of all '1's, except for the MSb, which is '0' (i.e., negative signed integers have the MSb equal to '1'). Therefore, the maximum value for a signed 16-bit integer is 32767. If the sum of two positive signed integers exceeds this maximum value, an overflow will occur. In this case, the result will wrap around to a negative value (i.e., the MSb becomes '1'), causing incorrect results and, as a consequence, potential unexpected behavior in the running application. Dually, negative overflows occur when the addition of two large negative values leads to a positive sum. Similar considerations hold in the case of fixed-point formats, as the involved arithmetic is the same, except for multiplications where additional right shifts are necessary to adjust the point position in the computed products. For example, when multiplying two fixed-point numbers with a limited number of decimal places, the result may be too large to fit within the available digits, leading to an overflow.

In general, overflows can affect computing systems as they are forced to represent data with a finite number of bits. However, this issue is exacer-

bated in hardware accelerators that usually deal with integer data of reduced bitwidths (e.g., 8-bit operands). This issue has been addressed with three main approaches. First, overflows can be prevented at the application level, by designing algorithms that can be effectively executed without incurring numerical overflows. Yet, this strategy may not be a doable solution in a wide range of applications. A second and more common solution is to saturate addition and multiplication outputs to the maximum positive or negative representable values when the overflow flag is raised [112]. This approach requires minimal circuit and computing overheads, but can largely affect output quality, as I will show in Section 5.3.4. Finally, overflows can be prevented by using larger data types for intermediate results [113]. For example, the sum of two 8-bit integers will not raise any overflow if the output is represented as a 9-bit operand. Similarly, the product of two 8-bit integers is safe from overflows if 16-bit products are employed. Dually, this strategy can be also seen as a limitation of the actual bitwidth of input operands [98, 114]. In this way, overflows can be prevented by constraining inputs to fit lower bitwidths than what they could use, ultimately ensuring that the calculated outputs will not be affected by overflows. Despite the fact that overflows can be prevented by construction, this approach can result in output quality degradation if inputs have to be truncated to fit reduced bitwidths. Moreover, it may also lead to poor memory utilization, since a significant fraction of bits devoted to input operands must remain unused to avoid overflows.

5.3.2 Overflow-free arithmetic for in-memory computing

Herein, I present a new approach, based on a HW/SW co-design strategy, that my colleagues and I have developed to prevent overflow in CNN inferences accelerated using In-Memory Computing (IMC) devices. On the hardware side, I show that the required circuital overheads to implement the proposed strategy are minimal and do not penalize efficiency in inference workloads. From an application-level perspective, I instead present an inference workflow designed for the in-memory acceleration of convolutional and fully-connected layers that supports *overflow-free* MAC operations in the designed IMC architecture. Moreover, the novel proposed CNN quantization methodology improves the previous proposal presented in Section 5.2 by increasing the inference cycle-count reduction of the optimized workloads.

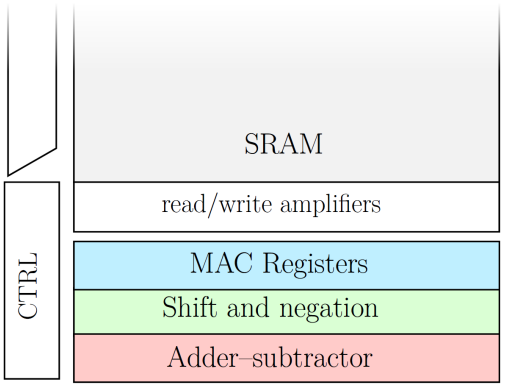


Figure 5.9: Overview of the IMC periphery, composed of the IMO pre-processing circuit (white box), registers for the MAC operations (blue box), in-situ shift and negation units (green box), and the adder (red box).

5.3.2.1 Architectural design

In-memory shift-add operations calculating multiplications are employed as described in Section 5.1.3, preventing overflow occurrences in the computed products. Instead, the IMC architecture must take special care when performing accumulations during MAC operations. To this end, the target IMC architecture has been enriched with MAC registers and a set of constant registers to manage overflows without significantly affecting performance and energy, and with a limited increase in area footprint. A conceptual representation of these additional elements in the periphery of IMC subarrays is illustrated in Figure 5.9.

The result of a product, stored in an accumulator register in the IMC periphery, is first added to the content of a new register, named MACL, using the in-memory adder. Note that this operation may induce positive or negative overflow. The type of overflow (i.e., positive or negative) is dictated by the MSb and carry-out bit of the calculated sum: when these are equal, no overflow occurs, while, if they differ, a positive (“01") or negative (“10") overflow occurs. Based on this observation, the proposed solution to manage overflows uses those bits to govern the state of a second register, named MACH. 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. A more detailed illustration of this register arrangement in the IMC periphery is shown in Figure 5.10. The MACH register is hence used as an overflow register, extending the bitwidth of the output products stored in MACL. Nevertheless, the output products stored in the register pair MACH-MACL must be cautiously manipulated. In fact, in CNN inference, the output activations of a certain layer become the input activations of the following one. In the previously discussed mapping strategy, activations of convolutional layers act as IMOs, which are then stored inside the IMC memory elements. For this reason, the computed outputs stored in two registers MACH-MACL have a bitwidth which is 2× than that of input IMOs. Therefore, these outputs are moved outside the IMC architecture where they are first post-scaled to balance the initial scaling of input IMOs. Moreover, before being loaded back into the IMC architecture for the execution of the second layer, they are also cast into 16-bit or 8-bit operands, according to the employed quantization level. In this sense, scaling and post-scaling operations are transparent from an accuracy perspective and only serve as a tool to constrain operands bitwidths inside specific boundaries.

5.3.2.2 Workflow for CNN acceleration

A novel workflow for the mapping of convolutional and fully-connected layers of CNNs considering the presented overflow-free IMC accelerator is depicted in Figure 5.11. The yellow box represents the considered IMC architecture, previously described in Section 5.3.2.1. I use it as a target hardware resource for the implementation of my proposed CNN optimization strategy, including input/output scaling and per-layer quantization strategies. Before being loaded in the IMC arrays, IMOs are scaled in the range $[0,1)$ either as 8-bit or 16-bit operands. As positive values, IMOs can be treated as *unsigned* operands, thus allowing their arithmetic precision to be increased by one bit (i.e., no need to store their sign bits). BOs are instead broadcasted as usual to compute dot products. Since the IMOs are scaled, the computed outputs must be de-scaled as well, to obtain the expected outputs. Both scaling and de-scaling operations, introduced to constrain operands into specific ranges,

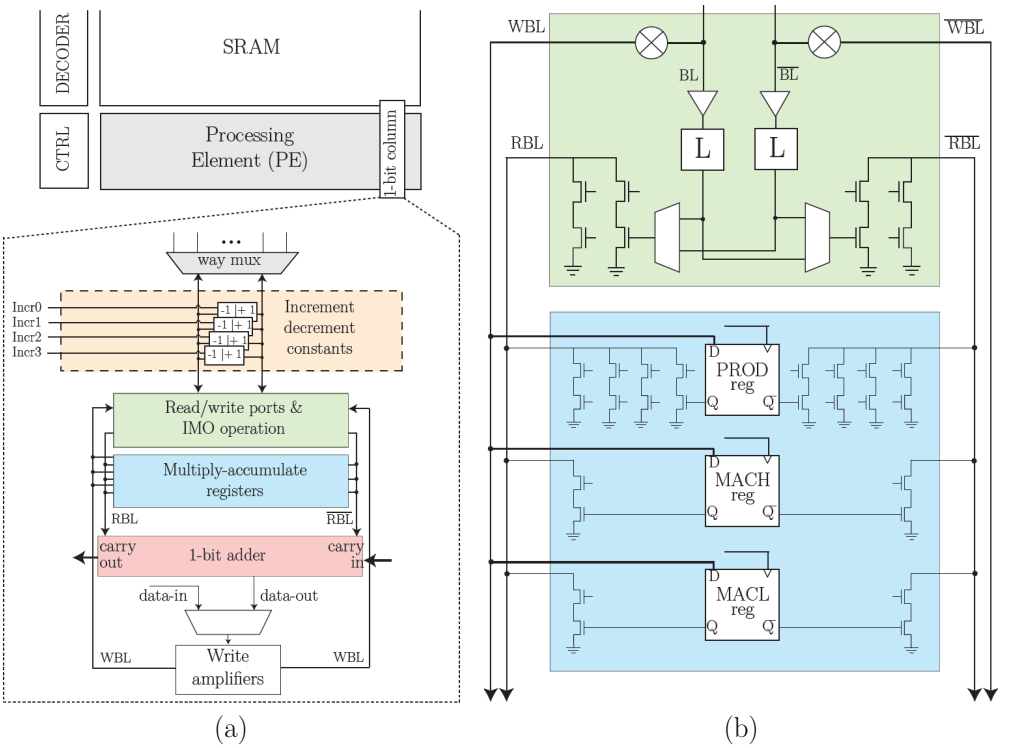
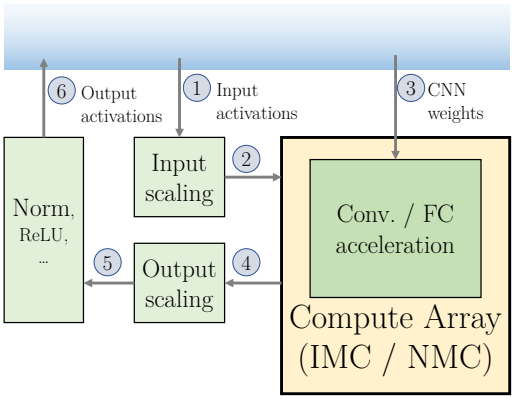


Figure 5.10: (a) Block diagram of the IMC architecture, highlighting a 1-bit column of the PE. (b) Detailed circuit implementation of the read ports and registers. $RBL \backslash \overline{RBL}$, and $WBL \backslash \overline{WBL}$ refer to the read and write path of the bit-lines, respectively.



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

Figure 5.11: In-Memory acceleration of CNN models. Input activations are first scaled (1) and then stored in the compute arrays as IMOs (2). Weights are instead broadcasted 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 on the right shows the data range at each step.

are performed outside the IMC architecture. This is also the case for the execution of other layers such as activation functions or normalization layers. Doing so, all these operations can be implemented using floating-point arithmetic, thus allowing the computation of precise scaling and normalization outputs. Moreover, performance is not significantly affected, as, for most CNN benchmarks, scaling and de-scaling operations, as well as ReLU and normalization layers account for less than 1% of the total inference workloads, which is indeed dominated by convolutional and fully-connected layers.

5.3.2.3 Improved heterogeneous quantization strategy

As in Section 5.2, CNN models must be optimized to fully leverage the efficiency benefits from the IMC acceleration. In a recent study [109], I have presented a novel heterogeneous compression methodology, based on a greedy strategy, to optimize the in-memory inference execution of CNN models. The proposed optimization is based on a per-layer quantization approach that improves the one I previously proposed in [101] (and illustrated in Figure 5.7). In fact, the limitation of the previous optimization method is that it

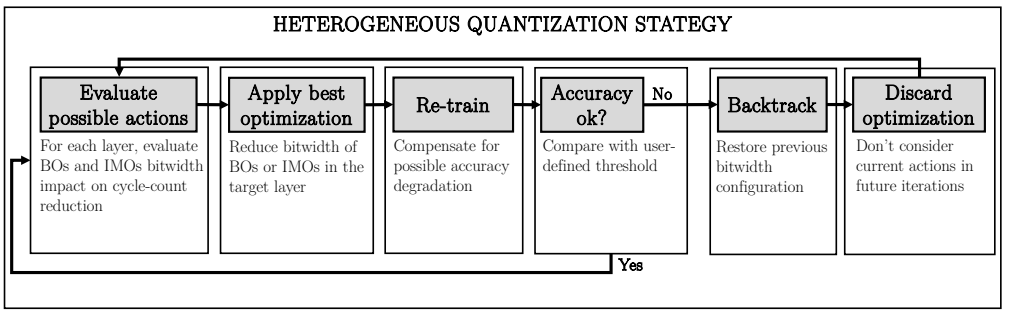


Figure 5.12: Heterogeneous CNN compression methodology. Among the possible quantization steps (weights or activations, in each layer), the one maximizing MAC cycle-count reduction is applied. The model is then re-trained and an accuracy evaluation determines if the imposed quantization scheme can be accepted or not. The optimization ends when no further optimization actions are possible.

targets BOs first, maximizing their bitwidth reduction by trading off the *whole* accuracy margin available. When no further bitwidth reductions are possible, the methodology aims to move IMOs from a 16-bit to an 8-bit quantization scheme, but having a very limited accuracy margin to exchange. The rationale behind that strategy was based on preliminary observations showing that IMOs in certain layers could be quantized more aggressively with no accuracy loss. Nevertheless, the reduction of IMOs bitwidth typically enables more aggressive cycle-count reductions by supporting word-level parallelism in dot product computation.

Therefore, I have proposed the newly refined approach summarized in Figure 5.12. This approach consists in a greedy optimization loop that aims at maximizing the cycle-count reduction of MAC operations at each step. To do so, it first evaluates the impact of bitwidth reductions of IMOs and BOs of different layers on the number of inference MAC cycles. In particular, it looks for the best optimization action among the $2N$ possible alternatives, considering CNN models with N layers (i.e., for each layer, individual optimization steps could either target IMOs or BOs). The selection of target operands of a specific layer is followed by their bitwidth reduction and, consequently, by a few re-training epochs to compensate for the possible accuracy drop.

Next, the accuracy of the CNN model is evaluated and compared with a user-defined accuracy constraint. The optimization loop continues if the achieved accuracy level abides by the user-defined constraint. Instead, if this is not the case, the previous quantization scheme is used for the optimized operands in the target layer. The corresponding optimization action is also discarded so that it will not be considered in future iterations. The optimization procedure ends when there are no more available optimization actions to be taken.

From a methodology 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 schemes. Therefore, considering a model with N layers, the complexity of the proposed optimization methodology can be approximated as $\mathcal{O}(8N)$, ultimately being linear in the number of layers. Notice that the optimization experiments conducted on the benchmarks proposed in Section 5.3.3.2 to evaluate this methodology take less than one day when performed on a Tesla V100 GPU. Conversely, an exhaustive exploration can evaluate all possible bitwidth combinations of IMOs and BOs of all layers. Still, although it returns optimal solutions, it is unfeasible in practice, as its complexity is exponential in the number of layers. In particular, it has a complexity of $\mathcal{O}((8\cdot2)^N)$, where 8 and 2 represent the number of possible BOs and IMOs bitwidths, respectively ($8\cdot2=16$ is the number of possible bitwidth combinations in *each* layer).

5.3.3 Experimental setup

5.3.3.1 Baselines

To evaluate the proposed co-design methodology comprising the CNN optimization strategy and the overflow-free IMC architecture from an accuracy, runtime, and energy perspective, we compare it with state-of-the-art overflow handling strategies. To perform a fair comparison, the considered solutions are all evaluated on the CNN benchmarks heterogeneously quantized according to the quantization method described in Section 5.3.2.3. Moreover, the considered baselines are directly applicable to the designed IMC architecture

without dedicated hardware support for overflow management. In other words, they represent state-of-the-art overflow-handling techniques that can be applied to the proposed IMC architecture, without relying on additional peripheral circuitry introduced in the considered architectural solution (e.g., MACH/MACL registers, constant registers, ...).

In a first baseline, overflow is managed by saturating large negative and positive values when overflow occurs [101, 112]. In particular, when an overflow is encountered, output activations are saturated to either the lowest or highest representable value of the employed quantized notation. I refer to this solution as *Saturation* when discussing the obtained results.

The second baseline prevents overflow by construction, ensuring that the accumulator register has a sufficient number of bits to obtain overflow-free accumulations of dot products in matrix-vector operations. To achieve so, the bitwidth of IMOs must be reduced. Considering subarrays composed of 256 rows and including 16-bit words, this overflow-preventing strategy must restrict the bitwidth of IMOs to only 8 bits. In fact, when performing dot products, the width required to perform the operation without any approximation and to avoid overflows is:

$$width(y) = width(x) + width(w) + log_2(V) \tag{5.1}$$

where y corresponds to the output dot product, x and w are activations and weights, respectively, and V is the number of added elements (i.e., the length of vectors x and w). Similar embodiments of this strategy are also proposed in several works on quantization [98, 113], not necessarily targeting in-memory computing architectures. In this baseline, referred to as *8-bit IMOs*, all IMOs are quantized as 8-bit values, but sign-extended to 16 bits, hence preventing the use of 2x8-bit IMOs configuration to parallelize computation (i.e., 8 MSbs in each memory word remain unused to prevent overflow in the accumulation registers).

5.3.3.2 Benchmarks

The following CNN models are considered in the conducted experiments: AlexNet [72], GoogLeNet [74], ResNet-8 [115], ResNext [84], VGG16 [83], and

Table 5.2: Baseline floating-point accuracy and complexity (memory and computing requirements) of the considered CNN models, evaluated on the CIFAR-100 dataset.

Benchmark	Accuracy (%)	Model size [MB]	GFLOPs
AlexNet	62.73	14.84	1.04
GoogLeNet	72.11	5.42	0.34
ResNet-8	59.53	5.88	0.19
ResNext	73.07	20.14	1.11
VGG16	60.39	14.62	0.29
MobileNet	47.19	3.15	0.32

MobileNet [76]. All models are evaluated on the CIFAR-100 dataset [73]. As a consequence, minor adjustments in the first layers are needed to adapt these models to the image sizes of the target dataset (i.e., 32x32 RGB inputs). As in previous chapters, the proposed benchmarks exhibit different degrees of complexity and are, hence, reasonable models to evaluate the proposed solution across a vast range of edge AI applications. They differ in the number of parameters, depth, and type of connections of the layers. Table 5.2 reports the achieved Top-1 accuracies, as well as the size and computational complexity of the presented benchmarks.

5.3.3.3 Pytorch-based environment for CNN training

CNN models are trained and quantized in a PyTorch-based environment [85], similar to the one employed for the experiments shown in previous chapters. First, the models are trained using floating-point precision until convergence. The BOs and IMOs are then quantized using 8-bit and 16-bit representations, respectively. As quantization may impact CNN accuracy, additional 20 re-training epochs are run, using fake-quantization [77] to force BOs and IMOs to assume quantized values. The re-training phase allows the 8/16 quantized benchmarks to recover in all cases from the accuracy drop due to quantization. Finally, benchmarks are optimized using the improved heterogeneous quantization methodology illustrated in Section 5.3.2.3, with an accuracy degradation constraint of 1% with respect to the one achieved by 8/16 quantized models.

5.3.3.4 Accuracy and runtime evaluations

The accuracy numbers presented in Table 5.2 are evaluated via a custom inference solver written in C++, extending the features of the one used for experiments shown in Chapter 2, as already described in 5.2.1. Indeed, it computes real fixed-point arithmetic and executes multiplication and accumulation operations as implemented in the compute arrays (illustrated in Figure 5.3) to accurately simulate the execution of the presented benchmarks. To this end, it also implements scaling and de-scaling functions, as described in the inference workflow depicted in Figure 5.11. In this way it allows me to estimate the impact of convolutional and fully-connected layers on the overall CNN inference runtime.

5.3.3.5 IMC implementation

The SRAM array is implemented as a full-custom design using a 28nm TSMC CMOS technology, implementing high-density memories using 6T SRAM cells. A varying number of 2KB subarrays (i.e., 4, 32, and 128 subarrays) are considered to evaluate the runtime improvements enabled by higher degrees of parallelism. Each subarray is organized as 1024 16-bit words. Four words in each memory row are bit-interleaved, so the implemented array presents 256 WLs and 64 BLs. This arrangement allows read and write accesses to operate at a limit of 2.2 GHz. Convolutional and fully-connected layers are executed and mapped on the IMC architecture on a layer basis. For each layer, the number of input and output words, and the number of cycles required to execute MAC operations is computed. Finally, as done in Section 5.2, IMOs are tiled to maximize compute parallelism while minimizing at the same time data movements.

5.3.4 Experimental results

5.3.4.1 Area and energy evaluation

An area and energy breakdown of the target IMC design as reported in [109] is presented in Table 5.3. The area of the SRAM subarray (i.e., the memory element, with no computing capabilities) is reported on the top-left side of the table and is used as a reference to evaluate the overhead of the IMC

Table 5.3: Area and energy breakdown of the SRAM subarray (256 WLs, 64 BLs) and the IMC Processing Element (PE).

SRAM	Area ($m\mu^2$)	Operation	Energy (fJ)
Bit-cell array	2129.9	Read	491.6
Read/write ports	81.0	Write	363.6
Word-line amplifiers	266.2	Leakage	88.9
Totals	2477.1	944.1	

IMC Processing Element	Area ($m\mu^2$)	Total Energy (fJ)
Adder	39.0	31.9
Shift/Negation	62.4	51.0
Registers	190.7	155.8
Totals	292.1	238.7

implementation. As expected, the largest fraction of the memory footprint (i.e., 85%) is due to bit-cells (2129.9μm^2 out of a total area of 2477.16μm^2), while read/write ports and word-line amplifiers have a minor impact on the total subarray size. The required additional area to implement in-memory computing capabilities is presented at the bottom-left side of the table. It is limited to 11.8% and is mainly due to the MAC registers illustrated in Figure 5.10. Instead, the 16-bit adder and shift/negation circuits have a minor impact on the total area overhead.

The right side of the table shows instead the energy breakdown of the SRAM subarray and the IMC processing elements. 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 leakage energy. The IMC computing capabilities consume a total of 238.7 fJ to perform shift-add operations, which represents a 50% energy reduction when compared to IMC operations performed with data accessed directly from the SRAM array, as shown in [107].

5.3.4.2 Improved heterogeneous quantization methodology

The bitwidth of IMOs and BOs of the CNN benchmarks optimized with the proposed heterogeneous methodology is illustrated in Figure 5.13. In particular, solid blue bars show the bitwidth of BOs in convolutional and fully-

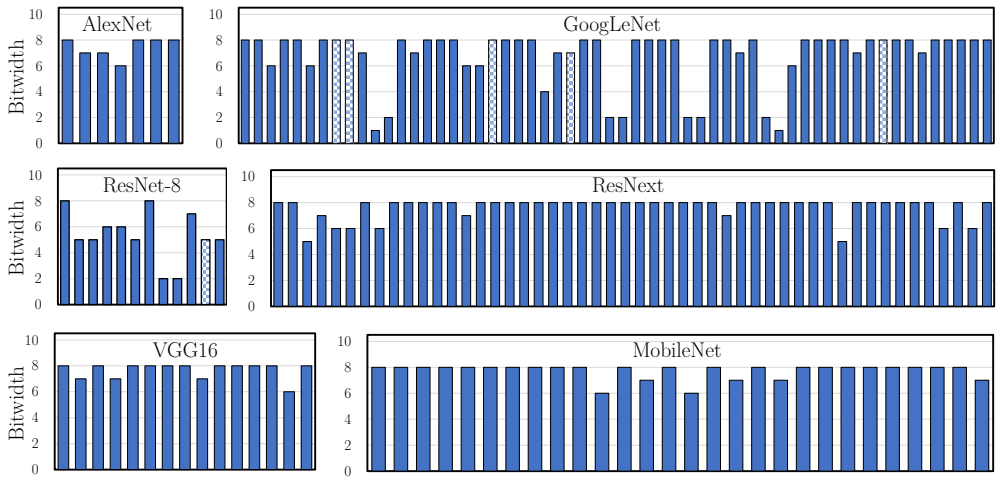


Figure 5.13: Per-layer bitwidth of BOs in the optimized benchmarks. IMOs are assumed to be reduced from the baseline 16-bit to 8-bit words, except for those (few) in layers in GoogLeNet and ResNet-8 marked with checkerboard backgrounds.

connected layers. IMOs are instead reduced to 8-bit words in all layers, except for those marked with a checkerboard background. 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. This is a major difference with respect to the results achieved with the previous strategy, where only a few layers have IMOs with reduced bitwidth. On average, this new optimization strategy reduces the number of shift-add clock cycles by 55.46% when compared to a baseline 8/16 quantized model, thus accelerating inference runtime and reducing energy. Moreover, compared to the initial optimization strategy presented in [101] and discussed in Section 5.2, this improved solution reduces the number of MAC cycles by 13% more, on average.

5.3.4.3 Comparison with baselines

Figure 5.14 illustrates the obtained inference runtime and accuracy results. In particular, the plots on the left of the figure show the inference runtime, expressed in seconds, of the optimized benchmarks executed in the designed IMC architecture. These results indicate that the proposed optimization

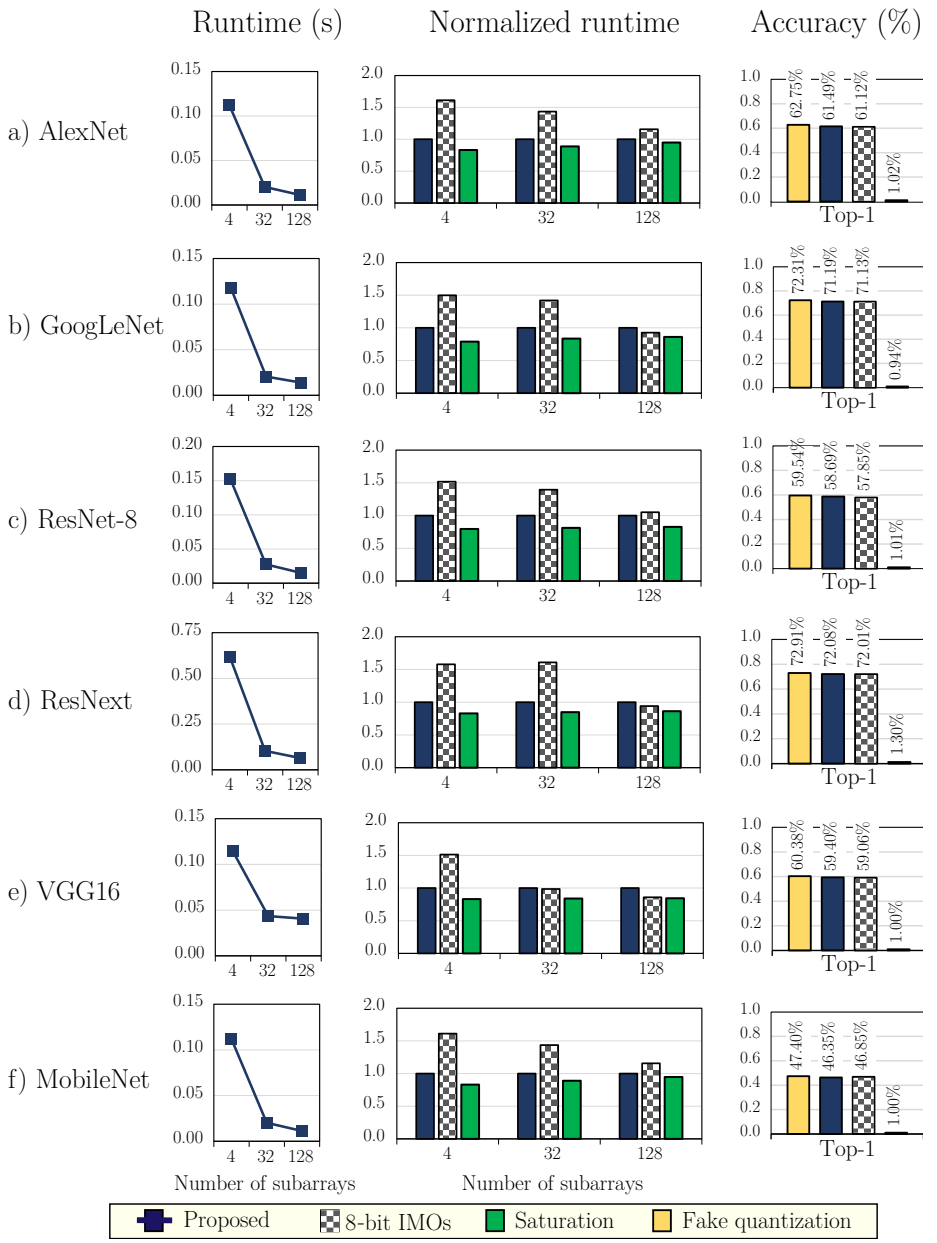


Figure 5.14: Inference runtime for different subarray arrangements (left). Normalized runtime of *8-bit IMOs* and *Saturation*, with respect to the proposed solution (center). Accuracy is reached with different overflow-handling strategies (right).

strategy scales nicely when increasing the number of subarrays. In fact, in comparison with a 4-subarray design, average inference speed-ups of 5.41× and 9.92× are measured in 32- and 128-subarray architectures, respectively. Nonetheless, diminishing performance gains are observed when moving from 32 to 128 subarrays. The reason is that convolutional and fully-connected layers are not fully exploiting the data parallelism offered by the hardware when too many subarrays are deployed (and when they are not large enough). Therefore, for edge AI applications where the number of convolutional filters is usually bounded to a few hundred, a large number of subarrays may lead to memory under-utilization.

The barplots at the center of Figure 5.14 show instead the inference runtime of the different overflow-handling techniques described in Section 5.3.3, normalized with respect to that of the proposed strategy. The runtime required for a single inference considering an increasing number of memory subarrays is reported. In this regard, the presented results assume that a unique BO is broadcasted to all the arrays at each IMC operation. Moreover, the results also assume both the *Saturation* and *8-bit IMOs* baselines require only one cycle for accumulations (since they only employ one MAC register), while the proposed solution requires two (to update MACL and also MACH). Average speed-ups of 60% are observed when comparing our solution with the *8-bit IMOs* baseline. The obtained performance improvements are mainly the result of the high degree of data parallelism enabled by our solution. Conversely, computing parallelism is prevented in the *8-bit IMOs* baseline, as the 8 MSBs of each 16-bit IMO must not be used to ensure overflow-free accumulations. The result is a non-optimal use of memory for the *8-bit IMOs* implementation, where 50% of memory words cannot be used for computation. Nevertheless, the *8-bit IMOs* approach results in slightly faster executions when considering IMC designs with 128 subarrays, 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 operations of our implementation). The proposed implementation results in an average 12% increase in inference runtime when compared to the *Saturation* baseline. One of the main reasons is that the runtime estimation for this second baseline considers saturation operations to have no impact on performance.

This assumption puts the *Saturation* baseline at an advantage because, in the proposed implementation, one additional IMC operation is required to update the value of the *MACH* register after each MAC operation.

Finally, the barplots on the right of Figure 5.14 depict the accuracy achieved by different approaches. The bars showing the accuracy achieved using a fake quantization strategy (yellow bars) are included to demonstrate that the proposed design can achieve comparable accuracy (blue bars), even when all intermediate operations abide by rigid bitwidth constraints. When comparing the proposed strategy with the *8-bit IMOs* baseline, the only (negligible) impact on accuracy is due to the shift-add implementation of multiply instructions, as overflow is prevented in both solutions. Instead, saturating the accumulator adversely affects accuracy, which drops dramatically in all cases for the *Saturation* baseline. Indeed, these experiments confirm that the number of saturated elements for each inference depends on multiple factors, including the CNN structure and the specific input image, and ranges from just 2% to more than 90% for certain images. On average, 37% of output activations are saturated across the evaluated benchmarks, thus motivating the very low accuracy of this second baseline.

5.4 Scaling SRAM voltage to improve efficiency

The key optimization objective discussed in this section is to increase even more the efficiency gains in IMC devices designed for accelerating CNN inferences. To achieve so, I herein present a methodology that enables the use of IMC accelerators at sub-nominal voltage levels to reduce energy consumption. The proposed approach combines together lightweight in-hardware parity check implementations, quantization, and the E^2CNNs approach described in Chapter 2, improving the robustness of CNN models against memory errors to support aggressive SRAM voltage scaling.

5.4.1 Operating SRAMs at sub-nominal voltages

Providing computing units and memory elements with sub-nominal voltage levels is a technique, known as voltage scaling, that reduces energy consumption. In fact, the energy cost of read and write accesses in an SRAM array

is directly proportional to the applied V_{dd}. Up to the point where leakage dominates, reducing the supply voltage levels in SRAMs significantly reduces energy without critically affecting performance. Therefore, the ability in reducing the energy expenditure of electronic systems makes voltage scaling a valuable solution in a wide range of low-power applications including edge AI, where energy efficiency is key. Voltage scaling can be either static or dynamic. In the first case (i.e., considered in this study), a sub-nominal voltage level is *continuously* supplying the memory device. In the second case, the voltage level is adjusted at runtime, with higher or lower voltages applied according to the application workloads or the actual battery level.

Yet, a sub-nominal input voltage in SRAMs may introduce permanent errors in the weakest memory cells, which become unable to flip their content. As discussed in Chapter 2, errors in memories can catastrophically affect the function of a system. AI workloads, such as the Convolutional Neural Networks (CNNs) I have considered in this thesis as evaluation benchmarks, are relatively resilient to errors, approximation, noise, and, more in general, data perturbations. In one of my first works, I studied with my colleagues the resiliency of an industrial CNN to memory stuck-at faults [60]. On one hand, we have demonstrated the importance of data representations, showing that compact fixed-point formats are more robust than floating-point ones. On the other hand, we have also observed that, if properly designed, the target CNN seemed to be quite resilient to errors up to certain error densities. Still, focusing on stuck-at-faults memory errors, different algorithmic and hardware strategies can be employed to increase the intrinsic CNN resiliency even more, opening the path to more aggressive voltage scaling strategies.

5.4.2 SRAM protection codes

Parity check is a simple and effective technique commonly used to detect errors in computer memories, ultimately ensuring data integrity. Although the focus of this section is on SRAM errors resulting from operating the memory system at sub-nominal voltages, memory errors can be generally due to noise, interference, or other factors. In all cases, these errors may cause the data to become corrupted, leading to incorrect results or system failures. To prevent such errors, error detection and error correction solutions

have been proposed. For example, the implementation of a parity check strategy allows to detect errors and only requires adding an extra bit to each data word. This extra bit is called the parity bit and is set to either '0' or '1' depending on the number of '1's in the data word. If the number of '1's is even, the parity bit is set to '0', while if it is odd, the parity bit is set to '1'. During data transmission or storage, the parity bit is also transmitted or stored along with the data word. When the data is accessed, the parity bit is checked to see if it matches the calculated parity bit based on the data word. A mismatch indicates that an error has occurred and the data word is corrupted. This memory protection approach demands minimal overhead and can detect single-bit errors in memory words. However, it cannot detect multiple-bit errors or errors that affect both the data word and the parity bit. Moreover, although it can detect single-bit errors, it is not able to *correct* the corresponding memory word, as it cannot say which bit is corrupted.

Error correction codes (ECCs) are a more advanced technique used in computer memories to detect and correct errors. To this end, ECCs use a more sophisticated algorithm than parity checks. ECC adds multiple redundant bits to each data word, which allows it to detect and correct single-bit errors. ECC is a valuable tool for improving system reliability and performance, especially in critical systems where data integrity is essential. However, it does require more memory and processing power than simple parity checks, which can make it more expensive to implement.

In the proposed approach illustrated in the next sections, I combine a lightweight parity check with an error mitigation strategy to reduce the effects of memory errors on the accuracy of CNN models.

5.4.3 IMC implementation of memory parity check

Herein, I will describe how the IMC architecture discussed in previous sections can be extended by including an error detection mechanism based on parity bit checks. In addition, I will also introduce a novel and simple mitigation strategy that reduces the negative effects of detected memory errors.

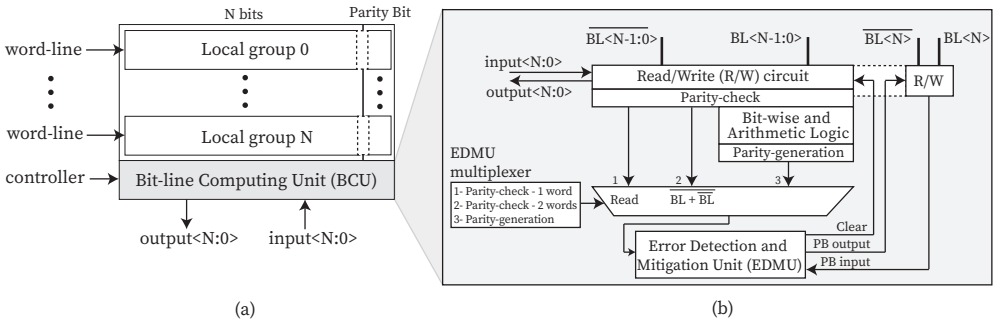


Figure 5.15: (a) In-memory computing architectural structure extending each memory word to allocate parity bits. (b) Peripheral circuit details, highlighting the Error Detection and Mitigation Unit (EDMU) used to compute parity checks and deal with detected memory errors.

5.4.3.1 Detection and mitigation strategy

To implement a parity check in the memory words of the IMC architecture described in previous sections, an additional bit must be included in each word-line, as shown in Figure 5.15-a. Moreover, additional circuitry, referred to as Bit-line Computing Unit (BCU) in the figure, is included in the array periphery. Indeed, in addition to the IMC capabilities already presented (e.g., multiplications, accumulations, and overflow management) the BCU must generate parity bits according to the stored memory words, implement parity checks, and take proper actions when errors are detected. Since these operations are implemented in the BCU, they are transparent from a system and application perspective and operate both during standard memory accesses and during single- or dual-operands in-memory operations.

The error mitigation strategy employed in this proposed design is extremely efficient yet effective, and aims at minimizing the circuit and computing overheads while reducing the accuracy impact of memory errors. When an error is detected in an accessed memory word during in-memory operations, its content is updated by filling all N bit-cells in the word line with '0'. Instead, in the case of standard memory accesses (i.e., simple read/write operations), this updated value is delivered to the memory output. Clearly, this approach cannot correct the memory word content and, in some extreme (and rare)

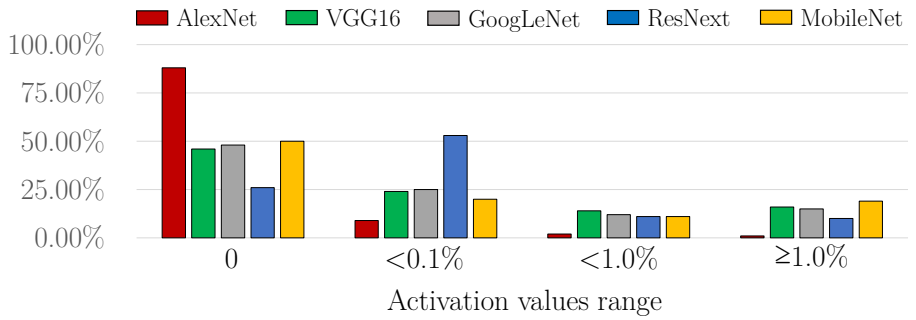


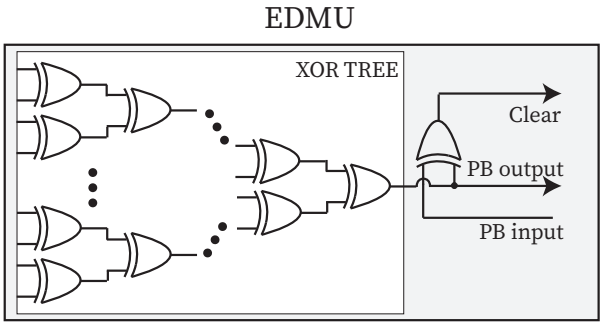
Figure 5.16: Activations distribution in the proposed benchmarks, illustrating the percentage of values being exactly zero, or lower than specific fractions of the entire representable range.

cases, it can introduce even large errors in the considered memory word. Nevertheless, the plots in Figure 5.16, showing the statistical distribution of activations in CNNs, motivate this approach. In different benchmarks, the distribution is highly skewed towards zero, with only a few outliers having a high magnitude. For example, in AlexNet (red bars), more than 80% of the activation values are 0, while 75% of them are smaller than 0.1% of the representable range in RexNext (blue bars).

The error detection process is also straightforward when only one operand is involved, requiring the XORing ($\oplus$) of all bit-lines, including that of the parity bit. When instead a two-operand in-memory operation is performed between two words $\mathbf{A} = \{A_{n-1}; A_{n-2}; \ldots; A_0\}$ and $\mathbf{B} = \{B_{n-1}; B_{n-2}; \ldots; B_0\}$, only the values $BL_i = A_i \cdot B_i$ and $\overline{BL_i} = \overline{A_i + B_i}$ are available, but not A_i and B_i themselves. Nonetheless, $A_i \oplus B_i$ can be computed as:

$$A_i \oplus B_i = \overline{BL_i + \overline{BL_i}} \tag{5.2}$$

As a consequence, parity checking can be performed using both bit-line signals, as follows:



PB input	PB output	Clear
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5.17: Error Detection and Mitigation Unit (EDMU) structure, composed of the XOR-tree and an additional XOR gate to compute the Clear signal.

$$\begin{aligned} Parity &= (A_{n-1} \oplus B_{n-1}) \oplus (A_{n-2} \oplus B_{n-2}) \oplus \cdots \oplus (A_0 \oplus B_0) \\ &= \overline{(BL_{n-1} + \overline{BL_{n-1}})} \oplus \overline{(BL_{n-2} + \overline{BL_{n-2}})} \oplus \cdots \oplus \overline{(BL_0 + \overline{BL_0})} \end{aligned} \tag{5.3}$$

5.4.3.2 Detection and mitigation circuit

As shown in [110], the IMC implementation of the presented mitigation strategy requires little hardware. A general structure of the peripheral circuitry implementing parity bit generation and parity checks is depicted in Figure 5.15-b, while the core architecture of the detection/mitigation circuitry, including a tree of XOR gates, named Error Detection and Mitigation Unit (EDMU), is illustrated in Figure 5.17. When a memory word is accessed, the EDMU detects if an error occurred and, if so, it assess the "Clear" signal to set to zero the value at the output of the read/write block. The EDMU inputs are either the bit-line signals (BL) for normal memory reads and single-operand in-memory operations, or the bitwise NOR between BL and $\overline{BL}$ signals for two-operands ones. The selection of proper inputs is dictated by a dedi-

cated multiplexer, itself governed by the memory controller illustrated in Figure 5.15-b.

During each in-memory operation, a new parity bit must be generated. To this end, the EDMU is traversed again, with the value computed by the arithmetic logic block (i.e., the output of a certain in-memory operation) at its input. The computed parity bit is then written back to memory simultaneously with the data bits of the computed memory word. All the actions outlined above can be executed in a single clock cycle. While the calculation of parity checks can be performed in parallel with in-memory arithmetic operations, the generation of a new parity bit does incur an additional (and marginal) delay of less than 10% of the circuit critical path, as evaluated in the experiments presented in Section 5.4.5.

5.4.4 Experimental Setup

Herein, I am going to present the experimental design used to evaluate the advocated HW-SW co-design methodology including (a) voltage scaling, to reduce energy consumption, (b) the in-hardware parity check mitigation strategy to reduce the impact of memory errors, and (c) E^2CNNs as an algorithmic-level transformation to support aggressive voltage reductions in SRAMs. Quantization is also employed in this methodology, but, for a fair comparison, it is assumed to be the baseline CNN implementation. In fact, I have already discussed how the findings I presented in [60] show that floating-point models are less resilient than quantized alternatives. Therefore, comparing the co-design approach introduced in this section with floating-point baseline implementations would not be sufficient to draw effective conclusions.

5.4.4.1 Single-instance and E^2CNNs benchmarks

The presented IMC architecture is evaluated on multiple CNN models including AlexNet [72], VGG16 [83], GoogLeNet [74], ResNext [84], and MobileNet [76], all tested on the CIFAR-100 dataset [73]. In addition to these single-instance architectures, E^2CNNs equivalent models including two and four instances are considered. As described in Chapter 2, these ensemble-based designs enable higher accuracy and robustness against memory errors,

without increasing the baseline memory and computing requirements. All CNN models are trained in PyTorch, using a fake-quantization approach [77] for the last 20 training epochs, using a uniform quantization level of 8-bit weights and 16-bit activations. Such a setting leads to negligible accuracy drops compared to floating-point implementations.

5.4.4.2 Stuck-at fault error model

Errors affecting SRAM are defined as faulty bit-cells permanently stuck at a given logic value, either '0' or '1'. In both cases, the designed model simulates the behavior of weak bit-cells, which are unable to switch their content during write operations as a result of insufficient current levels provided by sub-nominal input voltages. Due to the bit-cells inability to flip their content, these faults can be considered permanent errors, whose number and distribution in the SRAM array depend on the applied voltage level and on manufacturing process variability, respectively. During CNN simulations, the number of faults to inject in the emulated memory system is computed offline (i.e., before running the inference), according to the evaluated error rate. Subsequently, errors are injected as "stuck-at-1" or "stuck-at-0" faults directly in the selected bits of the memory array. Although the rate of soft errors producing temporary memory bit flips increases as well at low operating voltages, they are not considered in this analysis, because the considered (static) error rates are orders of magnitude higher (making the impact of soft errors negligible from an accuracy perspective).

As for the analysis in Chapter 2, the bit-flip probabilities reported in [51] for different supply voltage levels in $40nm$ technology are considered. The proposed error model introduces stuck-at faults in individual bit-cells by assuming, for each of them, a certain probability of being always set as a '1' or as a '0', irrespectively of its intended stored value. In some cases, the i_{th} bit of a memory word written in a memory word containing a stuck-at-fault in the bit-cell position i can match the value forced by the error itself. As a consequence, errors can be either observable or not observable. Faults cause observable errors if they affect the representation of the accessed data (e.g., a bit-cell stuck at '1' becomes observable when the desired value to be stored in that position is '0'). Assuming an equal probability of stuck-at-0 and

stuck-at-1 faults, the probability of having an observable error in a memory bit-cell during memory accesses is as follows:

$$P_e = \frac{1}{2} P_{stuck-at} \tag{5.4}$$

where $P_{stuck-at}$ is the probability of having a stuck-at fault in a bit-cell. Considering a word of N bits (possibly including a parity bit), the probability of having k bit-flips when accessing a memory word is then:

$$P_{(num-err==k)} = \binom{N}{k} P_e^k (1 - P_e)^{N-k} \tag{5.5}$$

Equation 5.5 is then employed to implement the developed error model as part of the C++ inference solver used in the experiments presented in this thesis. When executing multiply-accumulate (MAC) operations, a non-zero probability of bit-flips is assumed when computing output results. MAC operations are executed as a sequence of shift-adds among two operands (thus simulating real-hardware executions), in which each of the two may be affected by stuck-at faults. Without any error mitigation schemes, all bit-flips are propagated to successive computations. When instead simulating the proposed error mitigation strategy, output results are set to zero in the presence of an odd number of bit-flips (i.e., an even number of errors is not detectable by the implemented parity check). The probability of error detection is:

$$P_{error-detection} = \sum_{k=1,3,5,\ldots}^{k=N-1} \left[\binom{N}{k} P_e^k (1 - P_e)^{N-k} \right] \tag{5.6}$$

Table 5.4: Energy consumption per memory access and for each IMC operation, and bit error rate for an SRAM built on a 40 nm CMOS process at different voltage levels (fJ/bit) [51].

	Read	**Write**	**IMC op.**	**Error Rate**
800 mV	62.7	81.1	101.0	
750 mV	46.9	50.9	74.1	1×10^{-5}
700 mV	36.0	34.9	54.3	1×10^{-4}
650 mV	23.7	24.8	42.3	7×10^{-4}
600 mV	18.6	18.3	32.1	2×10^{-3}

Since the probability of having multiple errors (e.g., $k = 2,3,4,\ldots$) in the same accessed memory word decreases exponentially, only single-bit stuck-at faults in the SRAM words are considered in this model.

5.4.4.3 Energy and area evaluation

To be consistent with the error probabilities reported in [51], the in-memory computing architecture is implemented in 40nm CMOS technology. A summary of error densities at different error rates and the corresponding energy cost of read/write and in-memory operations is summarized in Table 5.4. As in the previously discussed IMC architecture designs, the memory is divided into multiple local groups including 16-bit words. An iso-size architecture not including the parity-bit memory cells and corresponding peripheral circuitry implementing the parity check and error mitigation strategy is also instantiated to evaluate energy cost in a baseline IMC design.

The number of read, write, and in-memory operations are assessed to retrieve energy requirements for different benchmarks. In this regard, the cycle-accurate simulator proposed in [101] (and also used in previous analyses in this chapter) is employed. For each CNN layer, the simulator computes the number of cycles required to load the inputs, perform the required in-memory operations, and stream out the results. Inputs are tiled, and channel-wise separation is performed if data size exceeds the available storage in the IMC memory array. In this case, additional operations are reported to stream

partial convolutions data to the memory array, perform the aggregation operations in memory, and retrieve the outputs.

5.4.5 Experiemental results

5.4.5.1 Area, energy and performance breakdown

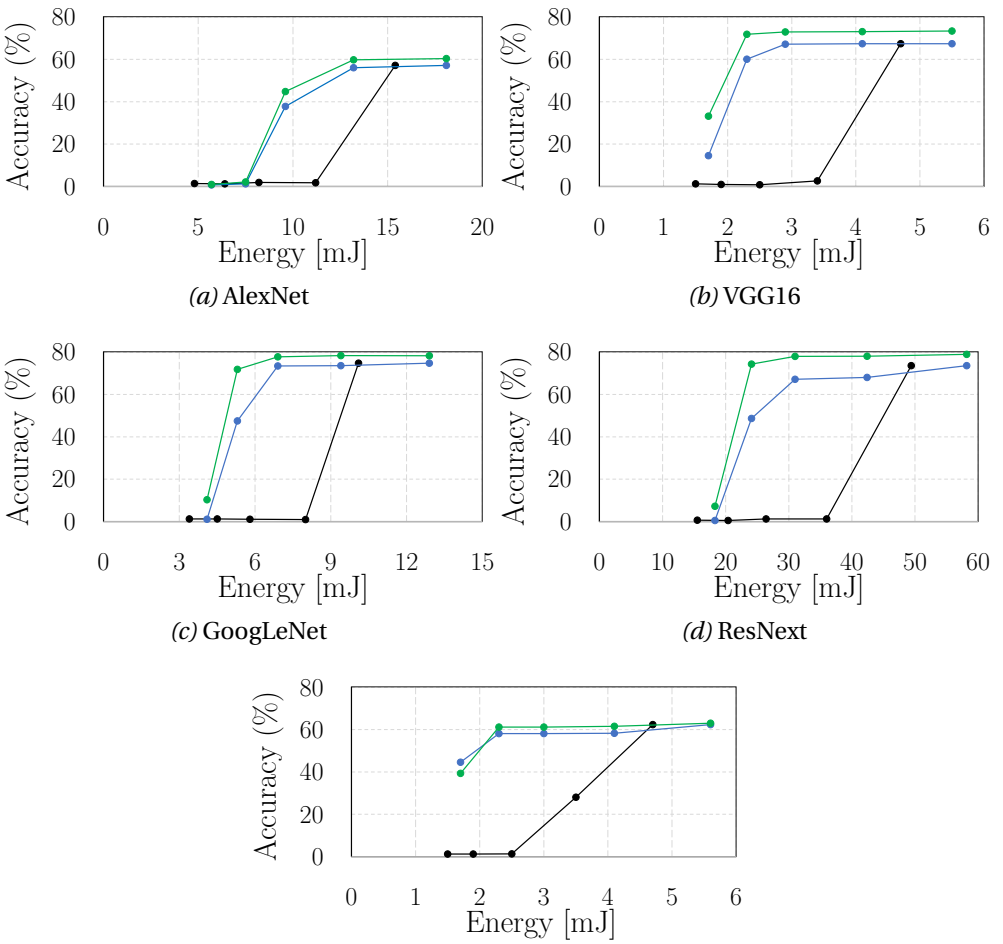
In the target IMC implementation, the majority of the area footprint (i.e., 76.9%) is occupied by the high-density SRAM bit-cells in the implemented IMC design. To this end, only 5.9% of this area is reserved for the parity bits, indicating a limited overhead for the storage of parity bits. The area overhead of the BCU supporting in-memory arithmetic operations and the proposed error mitigation strategy is also limited, and corresponds to 12.4% of the total area.

Per-bit energy values for read, write, and in-memory operations are included in Table 5.4. A supply voltage reduction from 800 mV to 700 mV reduces the energy of each operation by 47%, while at 600 mV, the total energy reduction reaches 72%. Still, the implementation of the error mitigation strategy increases reads and in-memory operations by 15%. This overhead is primarily due to the parity bit accesses and the EDMU, used twice during the same cycle in these operations, as described in Section 5.4.3.2.

Operating at sub-nominal voltages forces a reduction of the working frequency. Indeed, a supply voltage of just 650 mV reduces the maximum operating frequency by more than 40 % compared with an 800 mV baseline. Nevertheless, in this condition, the presented architecture is still able to run at 300 MHz, a relatively high operating frequency for ultra-low power edge AI applications. Importantly, in-memory operations are parallelized by employing multiple subarrays to increase throughput, which compensates for the frequency reduction.

5.4.5.2 Accuracy/energy trade-off

The accuracy achieved at different sub-nominal voltages is presented in Figure 5.18, where the energy cost of inference for different benchmarks is shown on the x-axis. Black curves correspond to baseline single-instance CNNs and



(a) AlexNet *(b)* VGG16 *(c)* GoogLeNet *(d)* ResNext *(e)* MobileNet

Figure 5.18: Inference energy reductions due to sub-nominal voltage memory operations produce errors that degrade accuracy. Black lines correspond to single-instance (quantized) baselines. The implementation of parity checks (blue lines) and, even more, its combination with E^2CNNs (green lines), better preserve accuracy while reducing energy.

blue ones represent the same models, in which the proposed error mitigation strategy is applied. Finally, green lines correspond to ensemble-based solutions composed of four instances (i.e., 4-E^{2}CNNs designs), also including the proposed error mitigation strategy. Markers from left to right correspond to increasing values of supply voltages as summarized in Table 5.4, from 600 mV up to the nominal 800 mV (a level in which no error occurs). These results indicate that voltage scaling dramatically impacts the accuracy of baseline CNN implementations. In particular, by slightly reducing the voltage from 800 mV to 750 mV the accuracy of the considered baseline models (black lines) is reduced by 59.8% on average.

On the other hand, Figure 5.18 also demonstrates the accuracy improvements of the proposed error mitigation approach at any evaluated sub-nominal voltage. More precisely, it can be observed that almost iso-accuracy executions are enabled in all benchmarks at 750 mV when implementing this mitigation strategy. Observing blue and black curves, it can be noticed that markers corresponding to the same voltage level are not vertically aligned to the ones of the single-instance baseline. The reason is that these results take into account the overhead of the error mitigation strategy implementation. Therefore, the energy cost for solutions implementing the proposed strategy is slightly higher for any supply voltage level. Still, the minimal energy overhead due to the additional circuitry performing the parity check is largely compensated by the possibility of reducing more the supply voltage, ultimately resulting in a more favorable accuracy/energy trade-off. On average, energy savings of 41.2 % can be achieved with the described methodology, while preserving the baseline accuracy. Also, extending the proposed HW-SW co-design approach with the described ensembles results in even more advantageous gains. In fact, the error mitigation strategy reduces the inexactness introduced by memory errors in the activations, thus limiting their impact on the accuracy of ensembles, that can achieve, on average, 8.2 % higher accuracy than single-instance CNNs. In this context, ensembles serve two objectives: on one side, they increase the initial inference accuracy at nominal voltage (i.e., error-free executions), and, on the other side, their additional robustness against errors is exploited to enable more aggressive voltage scaling. This effect is particularly evident in Figure 5.18 for VGG16,

GoogLeNet, and ResNext, where the curves of ensembles exhibit a smoother accuracy degradation than single-instance alternatives when the voltage is progressively reduced. Combining the presented error mitigation technique with ensemble-based solutions allows the supply voltage to be reduced to just 650 mV, resulting in energy savings up to 51.3 % with minimal impact on the initial CNNs accuracy in most benchmarks

Concluding remarks

6

This thesis has presented novel optimization approaches and methodologies to reduce the memory and computing requirements of CNN models, supporting their deployment and efficient execution in edge nodes. First, the proposed strategies consist of algorithmic-level optimizations that can be effectively leveraged in any sort of computing architecture. Then, to achieve higher efficiency levels, I illustrate hardware-software co-design methodologies showing a stronger design interconnection between application-level optimizations and hardware resources. In both cases, the main focus was on improving the inference execution of Convolutional Neural Networks (CNNs), deep learning models largely employed in multiple domains, from an energy-efficiency perspective.

The proposed E^2CNNs methodology has proved to be an effective algorithmic-level transformation to improve the accuracy and robustness of CNN models. It builds ensemble-based models that, in contrast to state-of-the-art ensembles, do not exhibit memory and computing overheads. In this regard, it combines pruning and replication, so that the ensemble instances are pruned versions of the original CNN implementation. To this end, experimental results revealed that its integration with other application-level optimization strategies, including quantization and codebook-based transformations, is beneficial to improve even more the inference efficiency and memory requirements.

For example, I applied the E^{2}CNNs methodology in conjunction with a codebook-based compression strategy that, targeting general-purpose hardware, reduces the memory requirements of AI workloads while improving performance at the same time. Compression is achieved by exploiting the limited size of codebooks, and small look-up tables storing a constrained number of unique model weights. In this way, the original CNN implementation is mapped as a set of codebook indexes (that require small bitwidths due to the reduced size of codebooks) accessing the corresponding weight values. Both the E^{2}CNNs methodology and the codebook-based compression approach are hardware-agnostic, in the sense that their benefits do not directly depend on the specific resources employed.

Instead, in the context of hardware-software co-design methods, I have shown how such strategies outperform alternatives focusing only on one of the two sides of the problem (i.e., either application-level optimizations or ad-hoc hardware design). To this end, I initially focused on hardware implementing inexact arithmetic, proposing an accuracy-driven methodology to map layers of CNN models into exact/inexact arithmetic resources, reducing energy while controlling accuracy to user-defined levels. Results have demonstrated that hardware-aware quantization approaches enable the use of specific multiplier units that, combined with a proper mapping approach, allow significant energy savings and controlled accuracy degradations.

I have also considered In-Memory Computing (IMC) accelerators, as devices offering even larger degrees of computing efficiency. In particular, I have developed a hardware-aware quantization strategy to effectively reduce the number of clock cycles in CNN inferences, ultimately improving performance and energy consumption. Moreover, to prevent arithmetic overflows in inference executions, I have proposed a workflow for the IMC acceleration of convolutional and fully-connected layers that includes scaling and de-scaling of IMOs. This method leverages the overflow registers instantiated in hardware, effectively preventing overflow occurrences by construction, without significant performance slowdowns. Finally, I have combined together E^{2}CNNs, quantization, and voltage scaling techniques in a more complete co-design methodology where a combination of algorithmic-level transformations and IMC hardware resources enable SRAMs to operate at subnominal voltage

levels, ultimately resulting in very high energy reductions, with limited or no impact on the output quality of CNN models.

I envision different extensions of the studies presented in this thesis. I briefly summarize them in the following section, separating them into short-term and long-term projects, based on the complexity and depth of possible investigations.

6.1 Possible extensions of proposed approaches

6.1.1 Short-term

- **Combined training of E^{2}CNNs instances.** As presented in Chapter 2, E^{2}CNNs is constructed by pruning an initial single-instance CNN model, to obtain pruned CNN structures that, once trained, will compose the designed ensemble. Still, each (replicated) pruned CNN instance is trained *independently* on the target training dataset. By randomly initiating weights in each instance, the training process generates CNN models with slightly different weight values, crucial to achieving higher accuracy and resiliency. Nevertheless, the training process could be potentially improved, by jointly training the generated CNN instances. In fact, different training strategies can be developed in this regard, either considering the E^{2}CNNs design as a whole intrinsic architecture (i.e., to backpropagate the gradients from the output of the ensemble to the N individual instances) or by employing orthogonal ensembling approaches such as bagging or boosting.

- **E^{2}CNNs of non-iso structured CNN models.** Another path to improve the E^{2}CNNs methodology is to avoid the restriction of having CNN instances with the same structure. In fact, by allowing individual CNN architectures to exhibit different structures, more heterogeneous and complex features could be extracted from input data, ultimately increasing the accuracy of the ensemble. In this sense, a lightweight meta-learner model [67] could be also included in the last stage of the E^{2}CNNs architecture to combine together individual predictions to produce more accurate estimations.

- **Quantizing codebooks for integer-only arithmetic.** Clustered weights stored in codebooks are floating-point values (see Chapter 3). On the one hand, this allows the execution of high-precision floating-point arithmetic in CNN inferences and, on the other hand, the proposed methodology effectively compresses CNN models without the need for specialized hardware resources. However, a slight variation of this methodology may be a valid solution for the hardware acceleration of CNN inferences. In this case, quantizing the codebook values to precise integer or fixed-point formats can enable efficiency gain in a wide spectrum of accelerators. However, the accuracy degradation due to quantization must be evaluated. As the generation of codebooks already produces several degrees of approximation, quantization should be carefully tailored. This phase should probably be included in the optimization loop when codebooks are actually generated. By doing so, accuracy can be effectively controlled, adjusting codebooks size and quantization levels while meeting the user-defined output quality requirements.

6.1.2 Long-term

- **E^{2}CNNs in federated learning.** Federated learning is emerging as an approach to train global models in distributed environments, abiding by privacy, latency, and data heterogeneity constraints [116]. In federated learning, a global AI model, initially trained in a centralized server, is then instantiated in a set of end devices. There, these copies are increasingly fine-tuned on the collected local data. Finally, periodic steps aggregate the updated local models to generate a new (and more accurate) global model on the centralized server. In this framework, E^{2}CNNs can be employed as an alternative to single-instance models. In addition to the improvements described in this thesis, E^{2}CNNs may offer additional benefits in the context of federated learning. For example, a certain number of instances of E^{2}CNNs in a local device can be kept *local*, undergoing the re-training stage, but avoiding being shared to update the centralized model. In this way, some instances can learn local features that characterize the data collected by the specific device. Additionally, since they are kept local, only the remaining instances

have to be shared for global updates, ultimately reducing latency and the energy cost for data transmission.

- **Approximate multipliers in hardware accelerators.** The last section of Chapter 4 underlined that the proposed methodology may offer appreciable energy efficiency gains when applied to highly-parallel computing units, citing systolic arrays as an example. An interesting research activity could be the design of systolic arrays employing exact and inexact multipliers, as discussed in this thesis. Then, the optimized benchmarks could be accelerated in the designed hardware resources, and the inference energy cost could be evaluated. Moreover, the use of systolic arrays and, more generally, the use of parallel computing resources open the door to much more complex exploration paths. For example, it can be investigated if all Processing Elements (PEs) can benefit from the use of approximate multipliers (in addition to the exact ones). Another option would be to use approximate multipliers in certain PEs only, a solution that would certainly demand a different CNN mapping strategy on the available resources. Finally, a third possibility would be to allow different degrees of approximation, designing some (of all) PEs with multiple approximate multipliers. The trade-off to be explored is still centered on performance, accuracy, energy, and area metrics, but shows a much higher degree of complexity with respect to the one presented in this thesis.

- **Full-system IMC simulation.** The presented co-design approaches on In-Memory Computing (IMC) focus on addressing specific challenges of the in-memory acceleration of convolutional and fully-connected layers of CNN models. Still, a future direction to develop this work could be to develop an overall framework and architecture integrating the IMC computing capabilities developed for SRAMs into an existing computing system. Recently, my colleagues and I have outlined such a comprehensive co-design vision in [117], describing a possible design solution that combines different optimization elements discussed in this thesis.

Bibliography

[1] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[2] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[4] Allen Newell and Herbert Simon. The logic theory machine–a complex information processing system. *IRE Transactions on information theory*, 2(3):61–79, 1956.

[5] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[6] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[7] TR Ramesh, Umesh Kumar Lilhore, M Poongodi, Sarita Simaiya, Amandeep Kaur, and Mounir Hamdi. Predictive analysis of heart diseases with machine learning approaches. *Malaysian Journal of Computer Science*, pages 132–148, 2022.

[8] Shristi Shakya Khanal, PWC Prasad, Abeer Alsadoon, and Angelika Maag. A systematic review: machine learning based recommendation systems for e-learning. *Education and Information Technologies*, 25:2635–2664, 2020.

[9] Johan Perols. Financial statement fraud detection: An analysis of statistical and machine learning algorithms. *Auditing: A Journal of Practice & Theory*, 30(2):19–50, 2011.

[10] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.

[11] Ali Bou Nassif, Ismail Shahin, Imtinan Attili, Mohammad Azzeh, and Khaled Shaalan. Speech recognition using deep neural networks: A systematic review. *IEEE access*, 7:19143–19165, 2019.

[12] KR1442 Chowdhary and KR Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pages 603–649, 2020.

[13] Arohan Ajit, Koustav Acharya, and Abhishek Samanta. A review of convolutional neural networks. In *2020 international conference on emerging trends in information technology and engineering (ic-ETITE)*, pages 1–5. IEEE, 2020.

[14] Jian-hua Li. Cyber security meets artificial intelligence: a survey. *Frontiers of Information Technology & Electronic Engineering*, 19(12):1462–1474, 2018.

[15] Zhaohui Liang, Andrew Powell, Ilker Ersoy, Mahdieh Poostchi, Kamolrat Silamut, Kannappan Palaniappan, Peng Guo, Md Amir Hossain, Antani Sameer, Richard James Maude, et al. Cnn-based image analysis for malaria diagnosis. In *2016 IEEE international conference on bioinformatics and biomedicine (BIBM)*, pages 493–496. IEEE, 2016.

[16] Debleena Paul, Gaurav Sanap, Snehal Shenoy, Dnyaneshwar Kalyane, Kiran Kalia, and Rakesh K Tekade. Artificial intelligence in drug discovery and development. *Drug discovery today*, 26(1):80, 2021.

[17] Kevin B Johnson, Wei-Qi Wei, Dilhan Weeraratne, Mark E Frisse, Karl Misulis, Kyu Rhee, Juan Zhao, and Jane L Snowdon. Precision medicine, ai, and the future of personalized health care. *Clinical and translational science*, 14(1):86–93, 2021.

[18] Ngozi Clara Eli-Chukwu. Applications of artificial intelligence in agriculture: A review. *Engineering, Technology & Applied Science Research*, 9(4):4377–4383, 2019.

[19] Christopher J Fluke and Colin Jacobs. Surveying the reach and maturity of machine learning and artificial intelligence in astronomy. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 10(2):e1349, 2020.

[20] Raquel Dias and Ali Torkamani. Artificial intelligence in clinical and genomic diagnostics. *Genome medicine*, 11(1):1–12, 2019.

[21] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[22] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[23] Jiang Wang, Yi Yang, Junhua Mao, Zhiheng Huang, Chang Huang, and Wei Xu. Cnn-rnn: A unified framework for multi-label image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2285–2294, 2016.

[24] Wang Zhiqiang and Liu Jun. A review of object detection based on convolutional neural network. In *2017 36th Chinese control conference (CCC)*, pages 11104–11109. IEEE, 2017.

[25] Meha Desai and Manan Shah. An anatomization on breast cancer detection and diagnosis employing multi-layer perceptron neural network (mlp) and convolutional neural network (cnn). *Clinical eHealth*, 4:1–11, 2021.

[26] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili,

Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

[27] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

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

[29] DT Mane and Uday V Kulkarni. A survey on supervised convolutional neural network and its major applications. In *Deep Learning and Neural Networks: Concepts, Methodologies, Tools, and Applications*, pages 1058–1071. IGI Global, 2020.

[30] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[31] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *2010 24th IEEE international conference on advanced information networking and applications*, pages 27–33. Ieee, 2010.

[32] Mark T Bohr and Ian A Young. Cmos scaling trends and beyond. *IEEE Micro*, 37(6):20–29, 2017.

[33] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[34] Yi Wei, Xinyu Pan, Hongwei Qin, Wanli Ouyang, and Junjie Yan. Quantization mimic: Towards very tiny cnn for object detection. In *Proceedings of the European conference on computer vision (ECCV)*, pages 267–283, 2018.

[35] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[36] Jian-Hao Luo and Jianxin Wu. An entropy-based pruning method for cnn compression. *arXiv preprint arXiv:1706.05791*, 2017.

[37] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[38] Jong Hun Lee, Joonho Kong, and Arslan Munir. Arithmetic coding-based 5-bit weight encoding and hardware decoder for cnn inference in edge devices. *IEEE Access*, 9:166736–166749, 2021.

[39] Akshay Jain, Pulkit Goel, Shivam Aggarwal, Alexander Fell, and Saket Anand. Symmetric k-means for deep neural network compression and hardware acceleration on fpgas. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):737–749, 2020.

[40] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. Pulp: A parallel ultra low power platform for next generation iot applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–39. IEEE Computer Society, 2015.

[41] Andrew Shell Waterman. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[42] Umair Saeed Solangi, Muhammad Ibtesam, Muhammad Adil Ansari, Jinuk Kim, and Sungju Park. Test architecture for systolic array of edge-based ai accelerator. *IEEE Access*, 9:96700–96710, 2021.

[43] Bai Sun, Tao Guo, Guangdong Zhou, Shubham Ranjan, Yixuan Jiao, Lan Wei, Y Norman Zhou, and Yimin A Wu. Synaptic devices based neuromorphic computing applications in artificial intelligence. *Materials Today Physics*, 18:100393, 2021.

[44] William Andrew Simon, Yasir Mahmood Qureshi, Alexandre Levisse, Marina Zapater, and David Atienza. Blade: A bitline accelerator for devices on the edge. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 207–212, 2019.

[45] Daniele Ielmini and Giacomo Pedretti. Device and circuit architectures for in-memory computing. *Advanced Intelligent Systems*, 2(7):2000040, 2020.

[46] Natalie Enright Jerger and Joshua San Miguel. Approximate computing. *IEEE Micro*, 38(4):8–10, 2018.

[47] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[48] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: Energy-efficient neuromorphic systems using approximate computing. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 27–32, 2014.

[49] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, 2012.

[50] Vojtech Mrazek, Radek Hrbacek, Zdenek Vasicek, and Lukas Sekanina. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 258–261. IEEE, 2017.

[51] Daniele Bortolotti, Hossein Mamaghanian, Andrea Bartolini, Maryam Ashouei, Jan Stuijt, David Atienza, Pierre Vandergheynst, and Luca Benini. Approximate compressed sensing: ultra-low power biosignal processing via aggressive voltage scaling on a hybrid memory multi-

core processor. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 45–50, 2014.

[52] Vishal Sharma, Ju-Eon Kim, Hyunjoon Kim, Lu Lu, and Tony Tae-Hyoung Kim. A reconfigurable 16kb and8t sram macro with improved linearity for multibit compute-in memory of artificial intelligence edge devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 12(2):522–535, 2022.

[53] Skanda Koppula, Lois Orosa, A Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 166–181, 2019.

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

[55] Issam Hammad and Kamal El-Sankary. Impact of approximate multipliers on vgg deep learning network. *IEEE Access*, 6:60438–60444, 2018.

[56] Flavio Ponzina, Miguel Peon-Quiros, Andreas Burg, and David Atienza. E 2 cnns: Ensembles of convolutional neural networks to improve robustness against memory errors in edge-computing devices. *IEEE Transactions on Computers*, 70(8):1199–1212, 2021.

[57] Marco Widmer, Andrea Bonetti, and Andreas Burg. Fpga-based emulation of embedded drams for statistical error resilience evaluation of approximate computing systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[58] David Middleton. Non-gaussian noise models in signal processing for

telecommunications: new methods an results for class a and class b noise models. *IEEE Transactions on Information Theory*, 45(4):1129–1149, 1999.

[59] Rahul Kher et al. Signal processing techniques for removing noise from ecg signals. *J. Biomed. Eng. Res*, 3(101):1–9, 2019.

[60] Benoît W Denkinger, Flavio Ponzina, Soumya S Basu, Andrea Bonetti, Szabolcs Balási, Martino Ruggiero, Miguel Peón-Quirós, Davide Rossi, Andreas Burg, and David Atienza. Impact of memory voltage scaling on accuracy and resilience of deep learning based edge devices. *IEEE Design & Test*, 37(2):84–92, 2019.

[61] Jinyu Li, Li Deng, Yifan Gong, and Reinhold Haeb-Umbach. An overview of noise-robust automatic speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(4):745–777, 2014.

[62] Michael L Seltzer, Dong Yu, and Yongqiang Wang. An investigation of deep neural networks for noise robust speech recognition. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 7398–7402. IEEE, 2013.

[63] Shuo-Yiin Chang and Nelson Morgan. Robust cnn-based speech recognition with gabor filter kernels. In *Fifteenth annual conference of the international speech communication association*, 2014.

[64] Mostafa Amin-Naji, Ali Aghagolzadeh, and Mehdi Ezoji. Ensemble of cnn for multi-focus image fusion. *Information fusion*, 51:201–214, 2019.

[65] Li Deng and John Platt. Ensemble deep learning for speech recognition. In *Proc. interspeech*, 2014.

[66] Bohdan Pavlyshenko. Using stacking approaches for machine learning models. In *2018 IEEE second international conference on data stream mining & processing (DSMP)*, pages 255–258. IEEE, 2018.

[67] Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple Classifier Systems: First International Workshop, MCS 2000 Cagliari, Italy, June 21–23, 2000 Proceedings 1*, pages 1–15. Springer, 2000.

[68] Pierrick Coupé, Boris Mansencal, Michaël Clément, Rémi Giraud, Baudouin Denis de Senneville, Vinh-Thong Ta, Vincent Lepetit, and José V Manjon. Assemblynet: A large ensemble of cnns for 3d whole brain mri segmentation. *NeuroImage*, 219:117026, 2020.

[69] Guanshuo Xu, Han-Zhou Wu, and Yun Q Shi. Ensemble of cnns for steganalysis: An empirical study. In *Proceedings of the 4th ACM Workshop on Information Hiding and Multimedia Security*, pages 103–107, 2016.

[70] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20(5):14, 2015.

[71] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

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

[73] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 55(5), 2014.

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

[75] Patrick Helber, Benjamin Bischke, Andreas Dengel, and Damian Borth. Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(7):2217–2226, 2019.

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

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

[78] Flavio Ponzina, Miguel Peon-Quiros, Giovanni Ansaloni, and David Atienza. An accuracy-driven compression methodology to derive efficient codebook-based cnns. In *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–6. IEEE, 2022.

[79] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[80] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. In *International Conference on Machine Learning*, pages 5363–5372. PMLR, 2018.

[81] Alfonso Rodríguez, Juan Valverde, Jorge Portilla, Andrés Otero, Teresa Riesgo, and Eduardo De la Torre. Fpga-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The artico3 framework. *Sensors*, 18(6):1877, 2018.

[82] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*,

2014.

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

[85] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[86] Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *JMLR. The Journal of machine Learning research*, 2011.

[87] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6. IEEE, 2013.

[88] Flavio Ponzina, Giovanni Ansaloni, Miguel Peón-Quirós, and David Atienza. Using algorithmic transformations and sensitivity analysis to unleash approximations in cnns at the edge. *Micromachines*, 13(7):1143, 2022.

[89] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

[90] Ilaria Scarabottolo, Giovanni Ansaloni, George A Constantinides, Laura Pozzi, and Sherief Reda. Approximate logic synthesis: A survey. *Proceedings of the IEEE*, 108(12):2195–2213, 2020.

[91] Giovanni Ansaloni, Ilaria Scarabottolo, and Laura Pozzi. Judiciously spreading approximation among arithmetic components with top-down inexact hardware design. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings 16*, pages 14–29. Springer, 2020.

[92] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[93] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.

[94] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. ACM, 2016.

[95] Pouya Houshmand, Giuseppe M Sarda, Vikram Jain, Kodai Ueyoshi, Ioannis A Papistas, Man Shi, Qilin Zheng, Debjyoti Bhattacharjee, Arindam Mallik, Peter Debacker, et al. Diana: An end-to-end hybrid digital and analog neural network soc for the edge. *IEEE Journal of Solid-State Circuits*, 58(1):203–215, 2022.

[96] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 3:25, 2019.

[97] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: a framework for bit-serial simd processing using dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–345, 2021.

[98] William Andrew Simon, Yasir Mahmood Qureshi, Marco Rios, Alexandre Levisse, Marina Zapater, and David Atienza. Blade: An in-cache

computing architecture for edge devices. *IEEE Transactions on Computers*, 69(9):1349–1363, 2020.

[99] Joshua Klein, Irem Boybat, Yasir Qureshi, Martino Dazzi, Alexandre Levisse, Giovanni Ansaloni, Marina Zapater, Abu Sebastian, and David Atienza. Alpine: Analog in-memory acceleration with tight processor integration for deep learning. *IEEE Transactions on Computers*, 2022.

[100] Mingu Kang, Sujan K Gonugondla, and Naresh R Shanbhag. Deep in-memory architectures in sram: An analog approach to approximate computing. *Proceedings of the IEEE*, 108(12):2251–2275, 2020.

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

[102] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(11):895–899, 2014.

[103] Daniele Ielmini and H-S Philip Wong. In-memory computing with resistive switching devices. *Nature electronics*, 1(6):333–343, 2018.

[104] Alexandre Levisse, B Giraud, JP Noel, M Moreau, JM Portal, et al. Architecture, design and technology guidelines for crosspoint memories. In *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 55–60. IEEE, 2017.

[105] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.

[106] Kaya Can Akyel, Henri-Pierre Charles, Julien Mottin, Bastien Giraud, Grégory Suraci, Sébastien Thuries, and Jean-Philippe Noel. Drc 2: Dynamically reconfigurable computing circuit based on memory archi-

tecture. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. IEEE, 2016.

[107] Marco Rios, Flavio Ponzina, Alexandre Levisse, Giovanni Ansaloni, and David Atienza. Bit-line computing for cnn accelerators co-design in edge ai inference. *IEEE Transactions on Emerging Topics in Computing*, 2023.

[108] Marco Rios, Flavio Ponzina, Giovanni Ansaloni, Alexandre Levisse, and David Atienza. Running efficiently cnns on the edge thanks to hybrid sram-rram in-memory computing. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1881–1886. IEEE, 2021.

[109] Flavio Ponzina, Marco Rios, Giovanni Ansaloni, Alexandre Levisse, and David Atienza. Overflow-free compute memories for edge ai acceleration. In *ACM Transactions on Embedded Computing Systems (TECS)*. ACM New York, NY, USA, 2023.

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

[111] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[112] Chun-Chieh Yang, Yi-Ru Chen, Hui-Hsin Liao, Yuan-Ming Chang, and Jenq-Kuen Lee. Auto-tuning fixed-point precision with tvm on risc-v packed simd extension. *ACM Transactions on Design Automation of Electronic Systems*, 28(3):1–21, 2023.

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

[114] Yesung Kang, Yoonho Park, Sunghoon Kim, Eunji Kwon, Taeho Lim, Sangyun Oh, Mingyu Woo, and Seokhyeong Kang. Analysis and solution of cnn accuracy reduction over channel loop tiling. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1091–1096. IEEE, 2020.

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

[116] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[117] Flavio Ponzina, Simone Machetti, Marco Rios, Benoît Walter Denkinger, Alexandre Levisse, Giovanni Ansaloni, Miguel Peón-Quirós, and David Atienza. A hardware/software co-design vision for deep learning at the edge. *IEEE Micro*, 42(6):48–54, 2022.

CURRICULUM VITAE

Ponzina Flavio

Personal Information

Chemin de l'Ormet 108, 1024 Ecublens (VD) - Switzerland
e-mail: flavio.ponzina@epfl.ch / flavioponzina@gmail.com
Born: 31/01/1994
Languages: Italian (native), English (fluent), French (B2)
Hobby/interests: track and field, modern physics, history and philosophy

Website: *https://people.epfl.ch/flavio.ponzina*
LinkedIn: *https://www.linkedin.com/in/flavio-ponzina-7b51a7139*
Publications: *Flavio Ponzina - Google Scholar*

Education

08/2019 – 09/2023
PhD student at Embedded Systems Laboratory (ESL)
Thesis – *Hardware-Software co-design Methodologies Edge AI Optimization.*

09/2016 – 12/2018
Master's Degree in Computer Engineering - *Embedded systems*
Thesis: *Hardware-Aware Optimization of Embedded CNNs.* - 110/110 *cum laude*

09/2013 – 07/2016
Bachelor's degree in Computer Engineering - 99/110

09/2008 – 07/2013
High school – Istituto tecnico industriale ITIS PININFARINA
Computer Science, Electronics - 100/100 *cum laude*

Work Experience

10/2023 – 09/2025
Post-doctoral researcher at University of California, San Diego (UCSD)
Large scale and in- and near-memory/storage processing systems

02/2019 – 07/2019
Internship with Embedded Systems Laboratory at EPFL.
Exploration and development of flexible, reusable and optimized hardware architectures to implement bio-signal processing and machine learning algorithms with ultra-low energy consumption

02/2017 – 04/2018
Software Engineer at PJM S.r.l.
Software development, network management, trainer of PHP/MySQL internal courses

09/2012 – 01/2017
ICT Consultant at MC TEAM.
Software development, feasibility studies, network management

06/2012 – 07/2012
Traineeship at MC TEAM.
Software development

List of Publications

- B. W. Denkinger *et al*., *"Impact of Memory Voltage Scaling on Accuracy and Resilience of Deep Learning Based Edge Devices*," *IEEE Design & Test*, vol. 37, no. 2, pp. 84-92, 2020.
- M. Rios *et al.*, "*Running Efficiently CNNs on the Edge Thanks to Hybrid SRAM-RRAM In-Memory Computing,*" *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, pp. 1881-1886, 2021.
- F. Ponzina *et al*, "*E2CNNs: Ensembles of Convolutional Neural Networks to Improve Robustness Against Memory Errors in Edge-Computing Devices*," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1199-1212, 2021
- F. Ponzina *et al.*, "*A Flexible In-Memory Computing Architecture for Heterogeneously Quantized CNNs*," 2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Tampa, FL, USA, 2021, pp. 164-169, 2021.
- F. Ponzina *et al.*, "*Using Algorithmic Transformations and Sensitivity Analysis to Unleash Approximations in CNNs at the Edge*," Micromachines (Basel), 2022.
- M. Rios *et al.*, "*Error Resilient In-Memory Computing Architecture for CNN Inference on the Edge*," Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22). Association for Computing Machinery, New York, NY, USA, 249–254, 2022.
- F. Ponzina *et al.*, "*An Accuracy-Driven Compression Methodology to Derive Efficient Codebook-Based CNNs*," *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, Barcelona, Spain, pp. 1-6, 2022.
- F. Ponzina *et al.*, "*A Hardware/Software Co-Design Vision for Deep Learning at the Edge*," IEEE Micro, vol. 42, no. 6, pp. 48-54, Nov.-Dec. 2022.
- M. Rios *et al.*, "*Bit-Line Computing for CNN Accelerators Co-Design in Edge AI Inference,"* IEEE Transactions on Emerging Topics in Computing, 2023.
- S. Zanoli et al., "*An error-based approximation sensing circuit for event-triggered, low power wearable sensors*," IEEE JETCAS 2023
- F. Ponzina et al., "*Overflow-free compute memories for edge ai acceleration*," ACM Transactions on Embedded Computing Systems (TECS), ACM New York, NY, USA, 2023.

Master's degree course projects

- Testing and fault tolerance - *"Development of a suite of software self-test procedures for the RI5CY pipelined processor"* – 2018
- Bioinformatics - *"Exploring SIMD vectorization for TGS algorithms"* – 2018
- System design project - *"IP-Core Manager for FPGA-based design"* – 2018
- Microelectronic Systems - *"Design and Development of DLX Microprocessors in VHDL"* – 2017
- Specification and simulation of digital systems - *"Design and simulation of a system on chip"* - 2017

Research domain/interests

- Hardware-software co-design optimizations
- Artificial intelligence, machine learning, deep learning
- AI for biomedical applications
- Edge AI
- Federated learning
- Hardware accelerators
- In-memory computing
- ML for space applications and Earth observation