Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Electrical, Electronics and Communications Engineering

(35^{th}cycle)

Techniques and Optimization Strategies for Efficient Hardware Acceleration of Neural Networks

Tap-Wisely-Quantized Winograd Algorithm and Capsule Networks

By

Beatrice Bussolino

******

Supervisor(s):

Prof. Maurizio Martina, Supervisor

Doctoral Examination Committee:

Prof. Francesca Palumbo. , Referee, Università degli Studi di Sassari

Prof. Stefania Perri, Referee, Università della Calabria

Dr. Renzo Andri, Computing Systems Lab, Huawei Zurich Research Center

Prof. Guido Masera, Politecnico di Torino

Dr. Riccardo Peloso, ST Microelectronics

Politecnico di Torino

2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Beatrice Bussolino
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Acknowledgements

I want to dedicate these lines to those who helped me to reach this goal and contributed to my journey.

I want to thank, first of all, Professor Maurizio Martina, for the opportunity he gave me and for his constant advice and support. I also would also like to express my gratitude to Dr. Renzo Andri and Dr. Lukas Cavigelli for mentoring me during the internship at the Zürich Huawei Research Center. Their supervision and collaboration have profoundly enriched my knowledge and skills.

Ai miei genitori, per il loro incondizionato sostegno e per l'incessante interesse dimostrato nei confronti del mio lavoro, malgrado la mia poca pazienza nello spiegare di cosa mi occupi. A Massimiliano, che da sempre alleggerisce le ore di studio e di lavoro, e che mi rende fiera ogni giorno. Ad Antonio, per aver arricchito questo dottorato e per essere diventato un punto di riferimento nel lavoro come nella vita, grazie alla sua straordinaria passione e dedizione. A Stefano, per aver sempre creduto nella nostra amicizia, da sempre faro nei momenti più bui. A Giorgio, per amarmi nonostante i miei limiti e per delicatamente spingermi sempre a superarli.

Abstract

The growing popularity of deep neural networks (DNNs) has intensified the demand for efficient hardware accelerators to handle the complex computations required by these models. This trend has led to increased research and development of domain-specific hardware accelerators (DSAs) to achieve the high performance and energy efficiency needed to support the deployment of DNNs in a wide range of applications. The effective execution of a DNN on a hardware accelerator depends on the workload presented by the model, the peak performance offered by the accelerator, and the efficiency with which the accelerator's resources are used.

With these three pillars in mind, in the first part of this thesis, we present a technique to enable convolutional neural networks (CNNs) acceleration by combining the Winograd algorithm for fast convolution and integer-only inference. A novel tap-wise quantization method overcomes the numerical issues arising when combining int-8 quantization and Winograd algorithm with larger tiles. Furthermore, custom hardware units and a carefully-tailored dataflow allow the processing of the Winograd transformations in a power- and area-efficient way. An extensive experimental evaluation on a large set of state-of-the-art computer vision benchmarks is conducted. This reveals that applying the tap-wisely-quantized Winograd algorithm with 4×4 tiles leads to a negligible accuracy loss compared to FP32 baselines. A domain-specific accelerator (DSA), enhanced with the Winograd custom hardware units, achieves up to $1.85\times$ gain in energy efficiency and up to $1.83\times$ end-to-end speed-up for state-of-the-art segmentation and detection networks.

In the second part of this thesis, we present our efforts to enhance the hardware-friendliness of capsule networks, a novel DNN model, by utilizing quantization methods and optimizing the model architecture in an HW-oriented fashion. This work aims to facilitate the acceleration of capsule networks, improving their computational efficiency and, in turn, enabling their deployment in resource-limited

environments. First, we present a study on the quantization possibilities for capsule networks and provide a framework for a fast generation of per-layer quantization parameters. When tested on a deep capsule network model for the CIFAR10 dataset, the proposed approach reduces the memory footprint by 6.2× with only a 0.15% accuracy loss. Secondly, we present NASCaps, an automated framework for the hardware-aware neural architecture search (NAS) of different types of DNNs, covering both traditional convolutional CNNs and capsule networks. The aim is to optimize the network accuracy and hardware efficiency, expressed in terms of energy, memory, and latency of a given hardware accelerator executing the DNN inference. The framework is tested on different datasets, generating various network configurations, and demonstrating the tradeoffs between the different output metrics.

Overall, this thesis presents novel techniques to overcome the challenges of CNNs and capsule networks and achieve efficient hardware acceleration. The results demonstrate that the proposed techniques improve the throughput and energy efficiency of the neural networks, which can have a significant impact on the development of efficient and accurate AI systems.

Contents

List of Figures

List of Tables

Acronyms

AIC AI core. 37

BatchNorm batch normalization. 5, 6

BLAS basic linear algebra subroutines. 9

CapsNet capsule network. 2, 7, 20, 21, 67–70, 72, 75, 76, 78, 80, 81, 119, 122, 123

CNN convolutional neural network. 1, 2, 4, 6, 7, 19, 20, 23, 27, 31, 32, 37, 38, 67, 70, 74–76, 119, 123

Conv convolutional. 4–7, 9, 11

DFG dataflow graph. 40

DL deep learning. 1, 9, 10

DNN deep neural network. 1, 2, 9, 14, 18, 19, 21, 68, 69, 94–97, 100, 103, 105, 106, 108–111, 113–119, 123

DSA domain-specific accelerator. 2, 20, 37, 65, 122

FC fully-connected. 3, 4, 6

FFT fast Fourier transform. 10

FGSM fast gradient sign method. 75

FIR finite input response. 24

FLOPS floating-point operations per second. 19, 94

FM feature map. 4

GPU graphics processing unit. 1, 7–9

HA-NAS hardware-aware NAS. 18, 19, 21

HW hardware. 2, 17, 121, 122

iFM input feature map. 4, 5, 10, 27, 34, 35, 38, 40, 45, 46, 48, 49, 58, 60, 64

IoT internet of things. 7, 119, 123

ISA instruction set architecture. 37

KD knowledge distillation. 14, 18, 36, 50, 51

MAC multiply–accumulate operation. 11, 12, 14, 20, 24, 29, 31, 35, 65, 68

MatMul matrix-matrix multiplication. 9, 14, 29, 36, 37, 39, 45–47, 63, 121

ML machine learning. 7–10

MTE memory transfer engine. 39

NAS neural architecture search. 7, 14, 16, 18–21, 75, 94, 107, 111–114, 123

NN neural network. 1, 7, 15–17

NoC network-on-chip. 10, 12, 13

NPU neural processing unit. 16

oFM output feature map. 5, 35, 40, 48, 56, 60, 64

PCC Person correlation coefficient. 105

PE processing element. 11–13, 16, 41–43, 46, 98–100

PGD projected gradient descent. 75, 112, 113, 115

PTQ post-training quantization. 15

QAT quantization-aware training. 15, 24

ReLU rectified linear unit. 3, 72, 98

RNS residue number system. 53

RTNE round to nereast even. 73, 74, 92

SQNR signal-to-quantization noise ratio. 84, 85, 90, 91

SR stochastic rounding. 73, 74, 92

TRN truncation. 73

VPU vector processing unit. 14

Chapter 1

Introduction

1.1 Context and Motivation

Deep neural networks (DNNs), born within the field of deep learning (DL), have seen significant growth and advancement in recent years, leading to their widespread adoption in a variety of fields including computer vision [6], natural language processing [7], language modeling [8], and even playing games such as chess and Go [9, 10]. The origins of DNNs can be traced back to the 1940s with Warren McCulloch and Walter Pitts' work on computational models of neurons. Still, it was not until the 1980s that DNNs drew more interest due to advances in computer hardware and the development of backpropagation, a training algorithm for DNNs [11]. However, only around 2010 DNNs began to see widespread success [12], thanks to the availability of large amounts of labeled data. One of the most significant milestones in the growth of DNNs was the development of convolutional neural networks (CNNs), leading to the achievement of state-of-the-art performance on tasks such as image classification [13].

One key challenge in the practical deployment of neural networks (NNs) is the high computational cost of training and inference. This cost can be mitigated through the use of hardware acceleration, which offloads computation to specialized hardware devices such as graphics processing units (GPUs). GPUs are the real workhorse for DL workloads in cloud infrastructures, thanks to their high parallelism and ability to perform matrix operations efficiently. However, in the last years, the growing popularity of DNNs but also the rising interest in edge computing for

real-time intelligent applications has led to increased research and development of domain-specific accelerators (DSAs) to achieve high performance and energy efficiency.

How effectively a DNN can be executed by an hardware (HW) accelerator depends on three factors [14]: the workload, the peak performance, and the efficiency. For the same accuracy, a simpler model with a lower **workload**, i.e., number of operations, always seems a favorable condition. For this reason, several efforts have been put toward developing small and efficient models [15–17]. However, when changing the workload, it is always crucial to consider the underlying hardware architecture to evaluate the effective benefits. For example, this is the case for depthwise convolutions, which cannot be efficiently accelerated by architectures working with channel-first data layout [18]. The **peak performance**, or throughput, of an accelerator, determines how fast a neural network can be processed. Increasing area and power consumption is a direct path to achieving higher peak performance, but it may not be sustainable in certain scenarios. However, it is possible to increase peak performance by acting at the algorithmic level, e.g., by simplifying the operations to be performed. Quantization is a commonly adopted technique, as moving from a 32-bit floating point to an 8-bit integer representation allows reducing area and energy costs and model size, while increasing the throughput. Finally, **efficiency** is determined by how well the computational resources are used, i.e., their utilization. To take full advantage of an accelerator, one must achieve maximum utilization of computational units, thus having a system that is always busy with computations (compute bound) and not blocked by memories (memory bound). This goal can only be achieved with a careful architecture design and dataflow mapping.

This thesis is based on the idea that all three aspects of workload, peak performance, and efficiency must be considered to effectively accelerate DNNs and are even more crucial when working with models that deviate from the well-established CNNs. The thesis is divided into two parts covering two different topics. The first part covers all three aspects of workload, peak performance, and efficiency when the Winograd algorithm for fast convolution is applied to CNNs. The second part shows how to increase the peak performance via quantization and careful model design of an innovative class of neural networks known as capsule networks (CapsNets).

The rest of the chapter is organized as follows. Section 1.2 introduces the background necessary to understand the presented works and pointers to state-of-

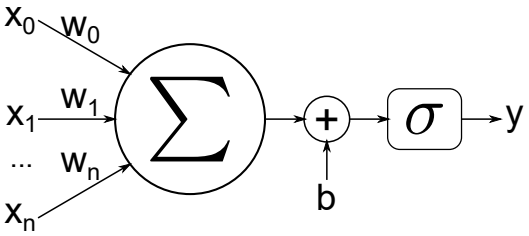
the-art research. Section 1.3 presents the main objectives and contributions of this dissertation.

1.2 Background and State of the Art

1.2.1 Deep Neural Networks

Neural Networks: Neurons and Layers

The basic element of neural networks is the *neuron*, a computational block conceived to model the behavior of biological neurons. It has been modified over time [19, 20] until reaching its modern shape, shown in Fig. 1.1. As described by Eq. (1.1), a neuron receives a set of inputs of which it computes a weighted sum. The output, also called *activation*, is then obtained by applying a non-linear function $\sigma(\cdot)$ to the weighted sum (Eq. (1.2)). The non-linear function makes neural networks non-linear systems, differentiating them from linear regression models. The most commonly used functions are the rectified linear unit (ReLU), the sigmoid, the hyperbolic tangent, and the softmax function.



$$g(\mathbf{x}) = \sum_{n=0}^{N-1} \mathbf{x}[n]\mathbf{w}[n] \tag{1.1}$$

$$y = \sigma\left(g(\mathbf{x}) + b\right) \tag{1.2}$$

Fig. 1.1 Model of an artificial neuron.

Neural networks are directed graphs with neurons as nodes. As shown in Fig. 1.2, the neurons of a neural network are organized in layers. Each layer receives a set of inputs from a previous layer and forwards its set of output activations to the following layer. The first neural network for handwritten digit recognition, LeNet [4], was composed of only five layers. Over time, thanks also to the increase of available computational resources, the number of layers of neural networks has grown, leading to the definition of *deep* neural networks [21].

The layers of a neural network can be differentiated by the connectivity they implement. The first layer to be introduced is the fully-connected (FC) layer. Given

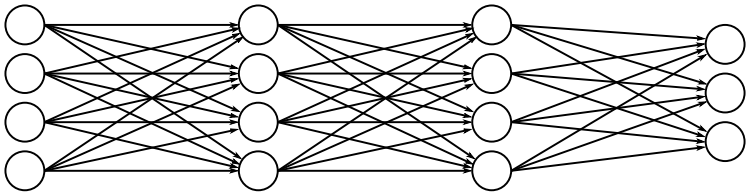


Fig. 1.2 Example of a generic neural network showing the disposition of neurons into layers

a set of M inputs and N neurons forming the layer, each of the neurons will receive and elaborate all of the M inputs. This can be expressed in equation form:

$$\begin{gathered}\mathbf{O}[n] = \sum_{m=0}^{M-1} \mathbf{W}[n,m]\mathbf{I}[m] + \mathbf{b}[n] \\ 0 \leq n < N, \;\; 0 \leq m < M\end{gathered} \tag{1.3}$$

From Eq. (1.3) we see that an FC layer can be written as a vector-matrix multiplication (Fig. 1.3), where the input vector is formed by the M inputs, and the matrix is built with the $M \times N$ weights of all the neurons.

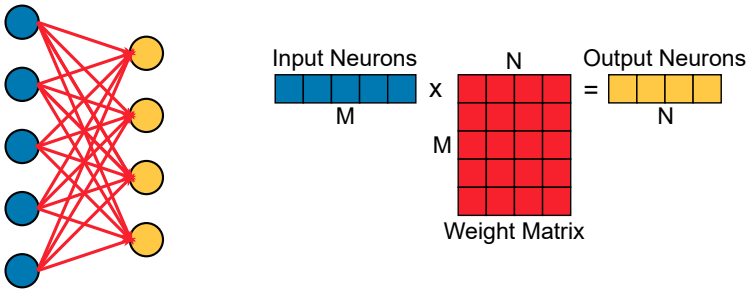


Fig. 1.3 Example of a fully-connected layer (left) and of how it can be modeled by a vector-matrix multiplication (right).

The number of inputs M and outputs N can be high, making the number of parameters of an FC layer potentially huge. Convolutional (Conv) layers alleviate this problem by exploiting the fact that a neuron does not necessarily need to work on all the outputs of the previous layer. Neural networks extensively adopting Conv layers are also called CNNs. CNNs are particularly suited for computer vision tasks, as Conv layers work on data arranged in a 2D grid, also called feature map (FM). A neuron of a Conv layer only receives information from a sub-grid of the whole input feature map (iFM), a receptive field of size $\langle H_k, W_k \rangle$. A single set of $H_k \cdot W_k$ weights is shared by all the neurons of a layer so that they are all detecting the same feature

but on different portions of the feature map. A Conv layer has multiple channels, specifically C_i input channels and C_o output channels, to detect multiple features. The computations performed in a Conv layer involve an iFM of size $\langle C_i, H_i, W_i \rangle$, the *weights* $\mathbf{W}$ of size $\langle C_i, C_o, H_k, W_k \rangle$, and a *bias term* b of size $\langle C_o \rangle$. The result of the computation is an output feature map (oFM) of size $\langle C_o, H_o, W_o \rangle$, computed as follows:

$$\begin{aligned} \mathbf{oFM}[c_o, h_o, w_o] = \sum_{c_i=0}^{C_i-1} \sum_{h_k=0}^{H_k-1} \sum_{w_k=0}^{W_k-1} \Big(\\ \mathbf{W}[c_i, c_o, h_k, w_k] \cdot \mathbf{iFM}[c_i, Sh_o + h_k, Sw_o + w_k] + \mathbf{b}[c_o] \Big) \\ 0 \leq c_o < C_o, \;\; 0 \leq h_o < H_o, \;\; 0 \leq w_o < W_o \\ 0 \leq h_k < H_k, \;\; 0 \leq w_k < W_k \end{aligned} \tag{1.4}$$

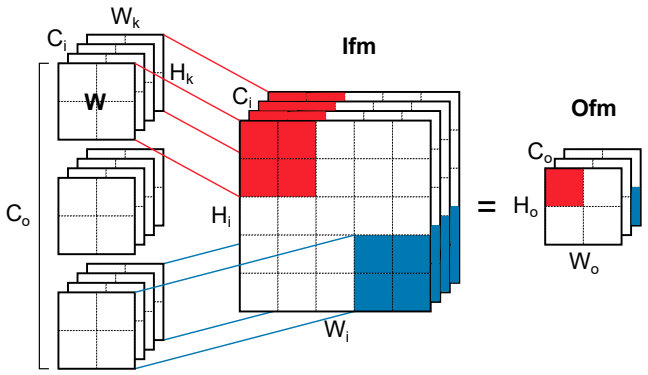


Fig. 1.4 Graphical representation of the convolution operation in a convolutional layer. A sub-tensor of **iFM** (red) is multiplied by a sub-tensor of **W** and the results are accumulated to produce a single value (red) of the **oFM**.

Conv layers are usually followed by a normalization operation that forces the layer outputs to have a normal distribution, i.e., zero mean and unit variance. This operation is beneficial because it maintains the numerical range constant for different inputs and avoids early saturation of activations, possibly caused by saturating non-linear functions like softmax or sigmoid. Moreover, normalization helps to speed up model training as the layers don't need to adapt to different distributions at each training step. The most popular normalization method is batch normalization (BatchNorm) [22], described by Eq. (1.5):

$$y = \frac{x - E[x]}{Var[x] + \varepsilon} \cdot \gamma + \beta \tag{1.5}$$

where $E[x]$ and $Var[x]$ are the mean and standard deviation of the input tensor x, respectively. ε is a value necessary for numerical stability, and γ and β are two trainable parameters for the integration of the BatchNorm layer in the training process.

Another type of layer commonly found in CNNs is the pooling layer, which is usually placed after a Conv layer. Pooling layers have receptive fields as Conv layers, but differently, their neurons do not perform a weighted sum of their inputs, but rather compute a statistic, such as the maximum or the average value. Pooling layers have the double function of reducing the number of activations of a layer, thus decreasing the number of computations of the following layers, and of making networks invariant to small local translations by downsampling.

Evolution of Neural Networks over the Years

Over the years, many CNN models were proposed achieving ever-improving accuracy.

For computer vision, as mentioned in the above paragraphs, the first popular CNN was LeNet [4], a model for handwritten digit recognition made of a simple sequence of two Conv and three FC layers only. Initially, the trend for CNNs was to stack more Conv layers: AlexNet [13], the first winner of the ImageNet Large Scale Visual Recognition Challenge 2012 (ILSVRC-2012) [23] with super-human accuracy on the ImageNet dataset [24], has the same structure of LeNet but with two Conv layers more. Samely, VGG [25], stemming from AlexNet, brings the number of layers up to 19.

Models such as ResNet [6] and GoogLeNet [26] further pushed the accuracy by creating blocks in which features obtained at different levels are concatenated or summed so that the following layers process richer information. Moreover, these solutions help with the *vanishing gradient problem* [27] that started to arise with the increasing number of layers stacked. Starting from the ResNet and GoogLeNet models, several other architectures were proposed [28–32].

Although (at least initially) increasing the number of layers seemed to be the easiest way to improve the accuracy, with the advent of internet of things (IoT) and edge processing, it also became necessary to focus on model efficiency. SqueezeNet [15] was the first small model to reach competitive accuracies by using more efficient blocks that reduce the network's overall number of parameters and computations. A similar idea is also present in MobileNet [16], which replaces expensive Conv layers with depthwise-separable Conv layers. EfficientNet [17] is a scalable network that allows the trade-off of computational intensity and accuracy, and that has outperformed all the older models on both fronts. MnasNet [33] is a model whose structure is obtained via neural architecture search (NAS) (see Section 1.2.3). The first popular network obtained with NAS is NASNet [34], and MnasNet differentiates from it as it is obtained by searching for both an accurate and efficient network.

Over time, architectures that differ from standard CNNs have been proposed. CapsNets [5, 2] were proposed to solve some of the problems of CNNs, such as the loss of data introduced by the pooling layers or the high sensitivity to inputs shifts or rotations (see Chapter 3). Recently, vision transformers emerged as a competitive alternative to CNNs. Transformers [7] are currently the most popular architecture for natural language processing and have been recently applied to computer vision tasks [35–37] with outstanding results in terms of accuracy and efficiency.

1.2.2 Hardware Acceleration of Deep Neural Networks

Many recent developments in the field of machine learning (ML) and NNs go hand in hand with advances in computational hardware, which have enabled these heavy algorithms by supplying the computational power needed to process massive amounts of data.

The most widely employed early architectures were general-purpose systems, such as CPUs and GPUs, with the latter being preferred especially because of their huge parallelism. However, many of the trends that sustained the advances in the performance of these architectures are ending, such as Moore's law or Denard's scaling. To further support the requirements of present and future ML algorithms, there has been an explosion of new architectures and computing technologies in recent years (see Fig. 1.5). These accelerators exploit a different trade-off between functional flexibility and performance w.r.t. general-purpose architectures. They

rely on specialized circuits for certain functions or operations characteristic of ML algorithms in particular.

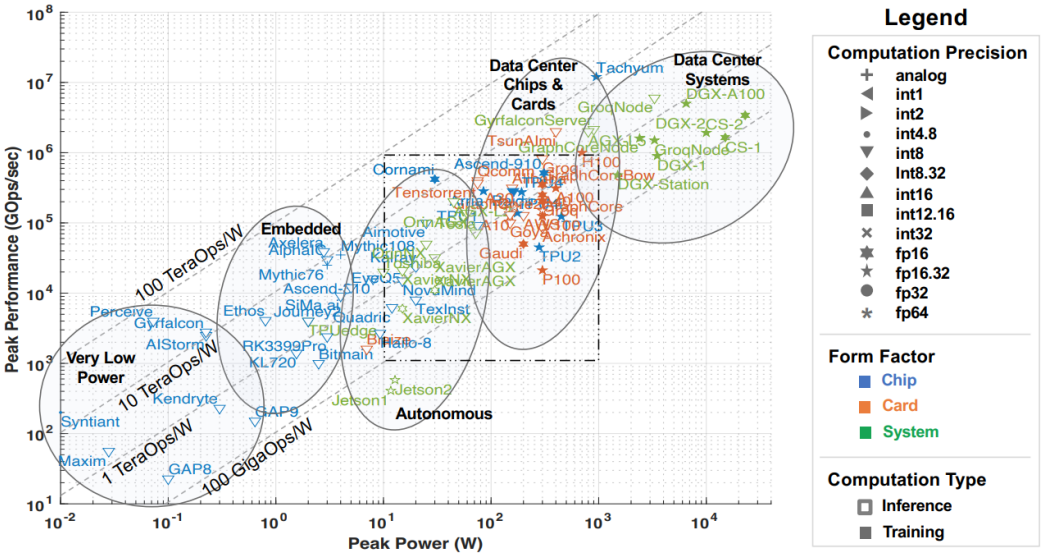


Fig. 1.5 Peak performance vs. peak power for publicly announced AI accelerators. Source [1]

In this crowded landscape, several classifications of the different architectures and systems are possible [1]. A reasonable classification is by the intended applications, which easily translates to power consumption, as shown in Fig. 1.5. On the plot, we see very-low-power chips, with sub-Watt power consumption, mainly used for edge computing nodes performing inference only; embedded solutions, with 1-10 W power consumption, for those applications requiring high-performance inference such as embedded cameras, small robots or UAV, etc.; chips with 10-100 W power consumption are mainly intended for inference only on autonomous systems; finally, above 100 W of power consumption, there are architectures for both inference and training in data centers.

The accelerators can then be classified by their architecture. For a long time, general-purpose architectures such as CPUs and GPUs have been used for ML workloads. Another family of architectures that is now very popular is that of dataflow processors, to which most of the specialized accelerators belong. Recently, accelerators based on processing-in-memory technology have been proposed and commercialized. All the solutions have in common that to accelerate ML algorithms efficiently, one must exploit their inherent parallelism, i.e., many operations can be executed in parallel since they do not have data dependencies.

Temporal architectures

CPUs and GPUs are general-purpose architectures that have also been classified as *temporal architectures* [38]. In temporal architectures, the computational resources can only access data from the central memory hierarchy, and the control is centralized. Both CPUs and GPUs allow exploiting data parallelism through single-instruction-multiple-data (SIMD) and single-instruction-multiple-threads (SIMT) execution models, respectively.

Despite these being general-purpose architectures, special attention has been paid to ML in recent years. At the software level, Intel has added the AVX-512 vector neural network instructions (AVX-512 VNNI) to the AVX-512 instruction set [39] to accelerate DNNs. Among the various GPU manufacturers, Nvidia has put a lot of emphasis on hardware and software optimization for DL. Most DL frameworks support the execution on Nvidia GPUs, e.g., Pytorch [40], Tensorflow [41], or Caffe [42]. One of the great advantages of Nvidia GPUs is cuDNN [43], a highly optimized library of primitives for DNNs. In the latest high-end GPUs, Nvidia has combined traditional CUDA Cores with Tensor Cores [44], which are optimized for large matrix operations. Tensor Cores can also support mixed-precision operations. In the new Nvidia A100, the Tensor Cores support a new format, the Tensor Format (TF32), with which performance is 10x higher when compared to the performance of the FP32 format on the V100 architecture [45]. In addition, Nvidia A100's Tensor Cores can also take advantage of the sparsity of tensors, very common in DNNs, to achieve up to 2x higher performance.

Several libraries are available optimizing basic linear algebra subroutines (BLAS) on both CPUs (e.g., AMD Core Math Library (ACML), Intel Math Kernel Library (Intel MKL) or OpenBLAS) and GPUs (e.g., Nvidia cuBLAS or Intel cIBLAS). Among the numerous subroutines implemented, the BLAS also include element-wise matrix multiplication, matrix-vector multiplication and matrix-matrix multiplication, also called general matrix-matrix multiplication (MatMul). For this reason, many of the DL libraries execute Conv layers by lowering them into a MatMul with the im2col algorithm [46, 47] as shown in Fig. 1.6. This method is very efficient since the MatMul routine is highly optimized. However, it requires data to be duplicated up to $H_k \cdot W_k$ times, with the dimension of the input feature maps moving from $\langle C_i, H_i, W_i \rangle$ to $\langle C_i H_k W_k, H_o W_o \rangle$. This approach, therefore, requires large memory for temporary allocation. The MatMul method for convolution can be furtherly

optimized by applying the Strassen algorithm [48, 49] that reduces the number of necessary multiplications by partitioning the matrices. The multiplications are reduced by 1/8 at each partition at the cost of more additions. A different approach consists of transforming both the iFMs and the weights from the space domain to the frequency domain with the fast Fourier transform (FFT) algorithm [50]. In the frequency domain, the convolution operation becomes an element-wise multiplication of matrices. However, the FFT algorithm introduces a high computational overhead for the domain change, and its efficiency has only been proven valid for large weight kernels and unitary strides. Another approach often used is based on the Winograd algorithm [51, 52] (see Chapter 2), which, unlike the FFT algorithm, is particularly efficient for small kernels.

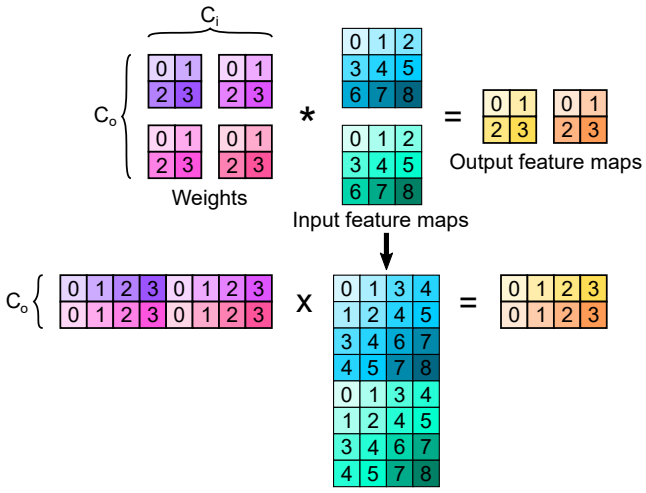


Fig. 1.6 Convolution lowering into a matrix-multiplication

Dataflow architectures

Dataflow processors are custom-designed architectures that accelerate ML workloads, leveraging that training and inference computations can be deterministically scheduled. These accelerators have also been defined *spatial architectures* [38], with computational resources that may have some local control logic and memory, and most importantly, can exchange data between each other via a network-on-chip (NoC).

As shown in Fig. 1.7, the general structure of most DL accelerators consists of a set of computational resources, on-chip memories, usually scratchpads, an NoC

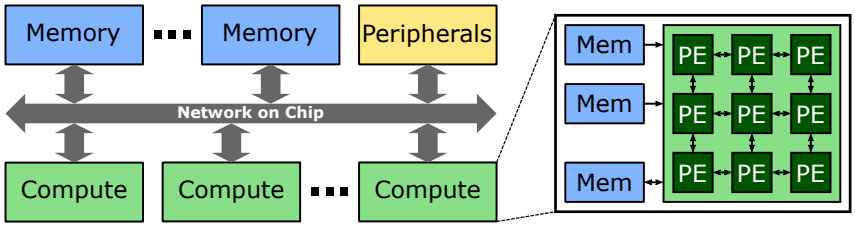


Fig. 1.7 General structure of an accelerator for AI applications.

to connect the various components, and the necessary peripherals to communicate with all off-chip resources, such as higher memory levels. Given the dominant presence of Conv layers in neural networks, much effort has been put towards their acceleration, with many hardware architectures proposed. The key operation of Conv layers is the multiply–accumulate operation (MAC). Thus many of the accelerators have as a basic component a processing element (PE) containing a multiplier and an accumulator, plus, depending on the specifics of the system, some additional control logic and registers for local storage.

In a convolution, every MAC operation requires three data elements from memory, an input pixel, a weight, and a partial sum for accumulation. It produces one result to be written back. Given the higher energy cost of a DRAM access w.r.t. a MAC [53], the DRAM access energy component can easily become the highest of the system [54, 55]. For this reason, accelerators are usually equipped with smaller on-chip memories, SRAMs or register files, and possibly registers close to the computational resources. Then, the mapping of the operations on the accelerator, or *dataflow*, revolves around *data reuse*, i.e., the scheduling of the operations to maximize the reuse of data stored in lower-level memories.

For example, a Conv layer has three different opportunities for data reuse. Since the weight kernel is slid over the whole input feature map, there is an opportunity for *weight reuse* since the same weight kernel is multiplied for multiple subsets of the input feature maps. In particular, each of the C_o kernels is reused $H_o \cdot W_o$ times. There is also an opportunity for an *input reuse* since the input feature maps are used C_o times to generate all the output feature maps. The last reuse opportunity is defined as *convolutional reuse* [56]. It exploits the sliding window mechanism, i.e., when computing two adjacent output pixels, there is usually an intersection between the two subsets of pixels of the input feature map used. The width and height of the intersection depend on the dimensions of the kernels (H_k, W_k) and the horizontal and

vertical strides (s_x, s_y). Convolutional reuse combines both weight reuse and input reuse.

Given an array of PEs and all the MACs between weights and input feature maps that must be performed to calculate the output feature maps, each PE will execute a subset of MACs, and a number of MACs equal to the number of PEs will be executed in parallel. The MACs must, therefore, be *spatially* and *temporally* mapped to the PEs array (Fig. 1.8). The mapping consequently defines how data must be loaded and stored from/to the memory hierarchy of the accelerator and how the NoC must be designed to correctly and efficiently deliver and collect the inputs, the weights, and the partial sums. Chen et al. [56] introduced a taxonomy to classify existing accelerators based on their dataflow and how they exploit data reuse, which will be explained briefly in the following.

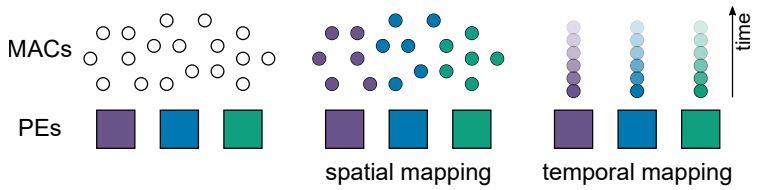


Fig. 1.8 Spatial and temporal mapping of the Multiply-and-Accumulate (MACs) operations to the Processing Elements (PEs).

The **weight stationary** dataflow aims at exploiting mainly the weight reuse: a subset of weights is read from the global memory (DRAM) and stored in the registers of the PEs. Possibly, all the operations that involve a certain weight are then mapped to the PE where it is stored, executed, and the weight can then be discarded and never reread from the DRAM. Considering a 2D array of PEs, some solutions [57–59] map the weights spatially on the PEs along the spatial dimensions of the weight kernel H_k and W_h, and others along dimensions C_i and C_o, as in the Tensor Processing Unit (TPU) [60] developed at Google. In the latter case, the operations that must be performed are equivalent to a vector-matrix multiplication, easily supported in hardware by a 2D systolic array. Because of its flexibility, the systolic array is often used in configurable architectures that must support various layer types [61, 62]. The **output stationary** dataflow minimizes the data movement necessary to store and load the partial sums in the global memory. The PEs are modified to have the possibility of locally accumulating the results of the MACs that they perform, and each PE is therefore responsible for the computations necessary to obtain an output pixel. To get an output stationary dataflow, it is possible to spatially map

the H_o and W_o dimensions [63], but also the H_k, W_k and C_o [64], or the H_o and C_o dimensions [65]. The **row stationary** dataflow is introduced in [56] and used by the Eyeriss accelerator [66]. It aims to maximize the reuse of inputs, weights, and partial sums altogether, in contrast to weight and output stationary dataflows focusing on a single data reuse type. The **no local reuse** dataflow [67, 54, 68] maximizes the area dedicated to storage by removing register files from the PEs and allocating all the on-chip memory in a single global buffer, in the view that the memory elements with higher energy efficiency are those with a low storage capacity, but they are less efficient in terms of area occupation (area/bit). Having no local reuse in the PEs, the traffic from and to the global memory on the NoC will be higher.

A critical aspect of the dataflow definition and accelerator design is the *loop tiling* technique. Usually, the on-chip memory size is insufficient to fully contain the input feature maps, kernel weights, and output feature maps. For this reason, it is necessary to partition the larger tensors into smaller tensors that can be contained in the on-chip memory. The **for** loops of the 7-nested loops representation of the convolutional layer are therefore split into multiple loops, as shown in Fig. 1.9. The tiling factors (TC_o, TC_i, TH_o, TW_o) define the size of the innermost loops and consequently the on-chip memory size required. In contrast, the permutations of the outermost loops determine the off-chip memory accesses and how the data are reused. Many levels of tiling can be applied, depending on the available number of levels in the memory hierarchy.

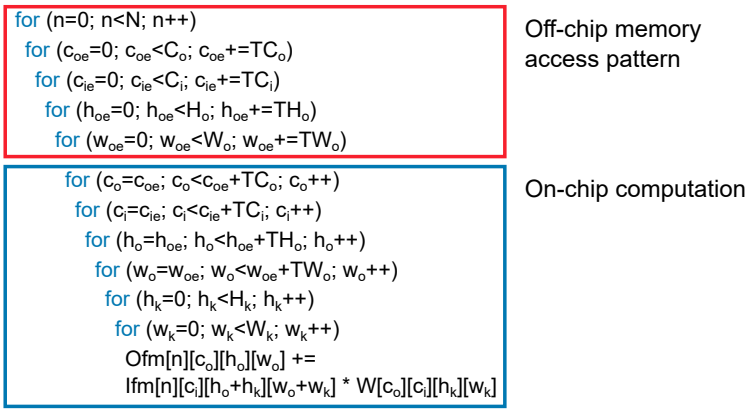


Fig. 1.9 Loop tiling technique applied to the 7-nested loops representation of the Conv layer

The described solutions are all very specific to convolution and matrix multiplication acceleration. However, in neural networks, many other operations are involved, for example, normalization or activation functions. For this reason, industrial-grade

accelerators often integrate other computational resources, usually more flexible, in addition to the main datapath used for convolution and matrix multiplication. This is the case for Google TPU [60], Huawei Ascend [69], and Qualcomm cloud AI 100 [70] platforms. These architectures are all characterized by a massive computational core for convolution/MatMul, a set of on-chip memories, and also integrate a vector processing unit (VPU) that can run activation functions, normalization, quantization, and all element-wise operations. To effectively accelerate a workload, it is very important to carefully plan the dataflow to achieve maximum utilization of the compute resources and, thus, maximum throughput.

1.2.3 Hardware-Efficient Deep Neural Networks

Two orthogonal approaches can be applied to increase peak performance and efficiency by acting at the algorithmic level. It is possible to modify an existing workload, e.g., with quantization, pruning, or knowledge distillation (KD), or to create a new model from scratch, e.g., with NAS. In this section, we briefly detail each of these methodologies.

Quantization

As discussed in the previous sections, one of the main obstacles to deploying DNNs on edge devices is their large memory footprint, the high energy cost of memory accesses, and the energy required for computations. Quantization is one of the most popular methods for reducing memory and computation requirements. This section provides the key concepts to understand the advantages and challenges of quantization and references to SOTA works. For more in-depth analysis, we refer the reader to [71].

Quantization is the process of mapping real values in floating point representation to a lower precision range. Fixed-point/integer representation allows for memory and energy saving, e.g., a MAC performed on 8-bit integer numbers consumes $20\times$ lower energy than a MAC on 32-bit floating-point numbers [53]. Moreover, a number expressed on 8 bits has a memory footprint $4\times$ smaller than one on 32-bits.

The most common and simple quantization function is defined by the following expression:

$$x_{\texttt{int}n} = \textbf{clamp}\left(\textbf{round}\left(\frac{x}{s}\right) - z, -2^{n-1}, 2^{n-1} - 1\right) \tag{1.6}$$

where s is a scaling factor, z is an integer zero point, and n is the chosen number of bits to represent the quantized value. The scaling factor s is computed as

$$s = \frac{\alpha - \beta}{2^n} \tag{1.7}$$

where α and β are the upper and lower bounds of the clipping range. If $\alpha = -\beta$ and $z = 0$, the quantization is symmetric. The most straightforward choice is to set the clipping range based on the min/max value of the signal, i.e., $\alpha = -\beta = max(|x_{min}|, |x_{max}|)$. Eq. (1.6) leads to uniform quantization, that is, the steps and levels of quantization are equispaced. Some works have also applied non-uniform quantization schemes [72, 73], but these techniques will not be further investigated here.

Methods to define the quantization parameters, i.e., the scaling factor and the zero point, can be divided into two categories, post-training quantization (PTQ) and quantization-aware training (QAT). PTQ is an easy solution as it does not require further training and can be applied with limited data. However, its effects on the accuracy degradations can be non-negligible. In PTQ [74–77], the weights and the activations are quantized after the training. This is easily done for the weights that are known and constant after the training, while for the activations a set of significative inputs is necessary for calibration.

On the other hand, in QAT [78–80], the weights are quantized during the training process so that the effect of quantization on the accuracy can be compensated. An example of this technique is provided by the Ristretto framework [79], which identifies the quantization parameters (bitwidth and scale factor) by running a statistical analysis on the weights and activations. Then, the weights are furtherly fine-tuned with a re-training step that takes quantization into account to recover the degraded NN precision. Some works [81] also consider the quantization parameters (zero-point, scaling factor) as learnable parameters and determine them at training time together with the values of the weights. Overall, QAT has proven the possibility of quantizing NNs with negligible accuracy loss. However, its main disadvantage is the higher computational cost, as it requires re-training the model for possibly several epochs.

Another important difference between quantization techniques is the granularity at which it is applied. A NN has several weight sets and feature maps. It is, therefore, possible to choose a single set of quantization parameters for the whole network. However, this is often not the optimal choice considering that the distribution of values can vary significantly throughout the model. Common options are *layerwise*, *groupwise*, *channelwise* and *subchannelwise* quantization. In layerwise quantization, quantization parameters are chosen independently for each layer's tensor (weight and activation). In channel-wise quantization, each channel of a tensor is quantized independently, while in group-wise quantization, different channels can be grouped to share quantization parameters. Finally, in sub-channel-wise quantization, values belonging to the same channel can be quantized separately. The higher the quantization granularity, the better different numerical distributions are captured, but also, the higher the computational overhead.

Several works have extensively demonstrated that neural networks can be quantized to 8-bit integer numbers for inference without significantly affecting the accuracy [79, 78, 82]. However, it is much more challenging to push the quantization to lower bitwidths if the bitwidth is maintained fixed across the model. *Mixed-precision* quantization addresses this problem. When also the bitwidth becomes a variable parameter, the search space for the quantization parameters fastly explodes. The space of possible solutions can be pruned based on heuristics [59, 83–85]. Several works studied how to automatize the search. For example, in [86], reinforcement learning is used to determine the optimal bitwidths per layer, while in [87], a NAS approach is used.

The research on fine-grained bitwidth optimization is backed by the parallel development of hardware accelerators that support flexible bitwidth arithmetic operations. Some accelerators support mixed-precision for the weights and/or the activations with bit-serial hardware [88–91]. In contrast, others have a spatial approach [92, 93], with PEs combined according to the required bitwidth. Mix-GEMM [94] exploits binary segmentation, a mathematical transformation. Tab. 1.1 reports a comparison between these different proposed solutions. On the industrial front, in 2018, Apple released the A12 Bionic chip with a neural processing unit (NPU) that supports variable precision; Nvidia Turing Tensor Cores, available in the Nvidia Turing architecture, support operations from 32/16-bit floating-point down to 8/4-bit fixed-point; the Imagination PowerVR Series2NX architecture has adjustable bitwidth from 16 to 4 bits.

Table 1.1 Comparison of different variable-bitwidth AI accelerators.

Name	Weights	Activations	Features	Target
BISMO [88]	1-bit to 8-bit	1-bit to 8-bit	Serial	FPGA
Stripes[89]	16-bit	1-bit to 16-bit	Serial	ASIC
UNPU [90]	1-bit to 16-bit	16-bit	Serial	ASIC
Loom [91]	1-bit to 16-bit	1-bit to 16-bit	Serial	ASIC
Bit Fusion [92]	1,2,4,8,16-bit	1,2,4,8,16-bit	Spatial	ASIC
BitBlade [93]	1,2,4,8,16-bit	1,2,4,8,16-bit	Spatial	ASIC
Mix-GEMM [94]	2-bit to 8-bit	2-bit to 8-bit	Binary segmentation	ASIC
Turing TC	64,32,16,8,4-bit	64,32,16,8,4-bit		GPU
PowerVR S2NX	16,8,4-bit	16,8,4-bit		SoC

Pruning

Given the redundancy of the parameters in NNs, pruning consists of removing, i.e., set to zero, those parameters that do not affect the accuracy of the model. Pruning was first explored in Optimal Brain Damage [95], where the weights with a lower influence on the loss function during the training are pruned. A simpler method [96] consists of pruning the weights with a small magnitude after the training and then fine-tuning the remaining weights to recover possible accuracy losses. Pruning single weights or neurons [97] is also referred to as unstructured pruning, oppositely to structured pruning that acts at kernel or channel level [98]. W.r.t. to structured pruning, unstructured pruning is more difficult to leverage to increase the throughput or reduce the energy consumption. In fact, unstructured pruning raises the sparsity of the computations, which requires dedicated strategies to be exploited. The pruning process has been applied jointly in several works [80, 99–102], and some solutions perform HW-aware pruning, where the pruning is guided by an estimate of the NN acceleration energy consumption [103, 104]

Knowledge distillation

Higher accuracies are obtained with very deep models or ensembles of models whose results are averaged. Using deep or several models at once requires considerable computational effort. However, it is possible to transfer the knowledge of one or more large models (teachers) into a smaller model (student). This process is commonly known as knowledge distillation and has been introduced in [105] and [106] for shallow and deep teacher models, respectively. In these works, the (trained) teacher models receive and classify a dataset of unlabeled data, producing a synthetically-labeled dataset. This dataset is then used to train the shallow student model, which,

therefore, learns to mimic the classifying function of the teachers. The KD method has shown promising results, and several variations have been proposed in subsequent works [107–110].

Architectural choices

A research direction is the exploration of new architectures with fewer parameters by construction. A basic idea is to replace a large kernel with a series of two or more smaller kernels. In this way, an equivalent receptive field is obtained but with fewer parameters. For example, a 5×5 kernel can be replaced by a series of two 3×3 kernels, reducing the number of weights from 25 to 18. In SqueezeNet [15], most of the 3×3 kernels are substituted with 1×1 kernels with $9\times$ fewer parameters, and the input channels to the 3×3 convolutions are reduced. SqueezeNet achieves the same accuracy as AlexNet with $50\times$ fewer parameters. In MobileNet [16], a standard convolution is divided into a depthwise convolution and a point-wise convolution. The depthwise convolution applies a different kernel to each input channel, while the point-wise convolution uses 1×1 kernels to combine the output channels of the depthwise convolution. This factorization reduces the number of parameters. Xception [111] adopts this same approach.

It is also possible to obtain smaller tensors from large tensors after training by applying tensor decomposition, a low-rank factorization technique. The kernels of the convolutional layers are 4D tensors, while the weights of the fully-connected layers are organized in a 2D matrix. With tensor decomposition, these can be broken down into tensors of lower dimensionality by canonical polyadic decomposition [112]. Since canonical polyadic decomposition is not numerically stable for tensors with more than two dimensions, it is possible to adopt Tucker decomposition [113].

Hardware-Aware Neural Architecture Search

Traditional NAS algorithms [114–116] have aimed at finding highly accurate DNN models for a given task, i.e., a DNN model which provides the highest accuracy on a given dataset. For example, the ENAS algorithm [114] has generated a new architecture with 55.6 perplexity on the Penn Treebank [117] dataset. Recently, the interest in hardware efficiency has been growing, leading to the design of hardware-aware NAS (HA-NAS) methodologies. The main difference between traditional NAS

and HA-NAS algorithms is that the latter also considers the hardware-deployment efficiency of candidate models, e.g., in terms of energy consumption, latency, or memory footprint. Among the related works, there exist mainly three types of heuristic search algorithms for the HA-NAS, which are (1) evolutionary algorithms, (2) reinforcement learning, and (3) differentiable NAS. APNAS [118], based on reinforcement learning, extends the ENAS algorithm by including the performance of DNNs executed in hardware in the optimization objectives of the NAS. AttentiveNAS [119] jointly optimizes the DNNs' accuracy and computational complexity expressed in floating-point operations per second (FLOPS). MnasNet [33] takes as an objective the inference latency and measures it by executing the candidate models on mobile phones. In [120], an extended search space is used, including architecture parameters, quantization, and hardware parameters, precisely the tiling factors. The FNAS algorithm [121] targets the FPGA and uses an analytical model to consider the latency only. HotNAS [122] targets energy efficiency by including model compression in the search space and supporting hardware for compressed models. SPOS [123] applies latency and FLOPS constraints during the candidate selection. HURRICANE [124] generates a search space tailored to a specific hardware platform, considering the FLOPS and the number of parameters and their effect on the latency. The Differentiable NAS (DNAS) framework [125], in which a stochastic super-net represents the search space, explores a layer-wise space where each layer of the CNN corresponds to a different block, and the learning is conducted by training the super-net.

1.3 Objectives and Contributions

This thesis explores different approaches to enable the hardware acceleration of neural networks, and the main objectives can be summarized as follows:

- Enable the acceleration of CNNs with the combination of the Winograd algorithm for fast convolution and 8-bit integer numerical representation, increasing throughput and energy efficiency.
- Study for the first time the quantization possibilities for capsule networks to facilitate their deployment in constrained environments and provide a framework for a fast generation of per-layer quantization parameters.

- Advance the research on capsule networks by applying neural architecture search to generate new, more accurate, and hardware-efficient models.

The thesis is divided into two main chapters, the first focusing on the quantized Winograd algorithm hardware integration and the second covering the quantization and NAS applied to capsule networks.

In Chapter 2, we show how to improve the throughput and energy efficiency of CNNs with the Winograd algorithm for fast convolutions, that lowers the number of MACs w.r.t. the standard algorithm, reducing the operation count by a factor of 2.25× for 3×3 convolutions when using the version with 2×2-sized tiles F_2. Even though the gain is significant, the Winograd algorithm with larger tile sizes, i.e., F_4, offers even more potential in improving throughput and energy efficiency, as it reduces the required MACs by 4×. Unfortunately, the Winograd algorithm with larger tile sizes introduces numerical issues that prevent its use on integer DSAs. It also has a higher computational overhead to transform input and output data between spatial and Winograd domains. To unlock the full potential of Winograd F_4, we propose a novel tap-wise quantization method that overcomes the numerical issues of using larger tiles, enabling integer-only inference. Moreover, we present custom hardware units that process the Winograd transformations in a power- and area-efficient way. We show how to integrate such custom modules in an industrial-grade, programmable DSA. An extensive experimental evaluation on a large set of state-of-the-art computer vision benchmarks reveals that the tap-wise quantization algorithm makes the quantized Winograd F_4 network almost as accurate as the FP32 baseline. The Winograd-enhanced DSA achieves up to 1.85× gain in energy efficiency and up to 1.83× end-to-end speed-up for state-of-the-art segmentation and detection networks.

In Chapter 3, we show a set of orthogonal techniques to be applied to capsule networks to address their present challenges. Capsule networks were proposed to solve some of the problems of CNNs. Still, they require highly intense computations and are difficult to be deployed in their original form on resource-constrained edge devices. For this reason, we present a specialized quantization framework for CapsNets to enable their efficient edge implementations. The framework is evaluated on several benchmarks. On a deep CapsNet model for the CIFAR10 dataset, the framework reduces the memory footprint by 6.2×, with only 0.15% accuracy loss. Another critical aspect in the research field of capsule networks is that, despite

their promising initial results, few new models have been studied and proposed in the literature. Thus, in the second part of Chapter 3, we propose NASCaps, an automated framework for the HA-NAS of different types of DNNs, covering both traditional convolutional DNNs and CapsNets. We study the efficacy of deploying a multi-objective genetic algorithm based on the NSGA-II algorithm. Considering the computational intensity of capsule networks, the proposed framework can jointly optimize the network accuracy and the corresponding hardware efficiency, expressed in terms of energy, memory, and latency of a given hardware accelerator executing the DNN inference. Besides supporting the traditional DNN layers (such as convolutional and fully-connected), our framework is the first to model and support the specialized capsule layers and dynamic routing in the NAS flow. We evaluate our framework on different datasets, generating various network configurations, and demonstrate the tradeoffs between the different output metrics. Moreover, we also show how the framework can easily be extended to incorporate other objectives in the search, e.g., robustness to adversarial attacks.

Finally, Chapter 4 summarizes the contributions discussed throughout the thesis.

Chapter 2

4x4-Tiles Winograd Convolutions: Quantization and Efficient Inference

The work presented in this chapter was developed in the context of an internship at the Huawei Zürich Research Center under the supervision of Dr. Renzo Andri and Dr. Lukas Cavigelli and appears in [126].

2.1 Introduction and Motivation

As thoroughly discussed in Chapter 1, floating-point representation is typically used for CNNs training and inference, even though floating-point datapaths are area- and power-hungry because of the hardware necessary, for example, for large intermediate values, exception handling, or re-normalization. Integer-based operations, on the other hand, have higher energy efficiency and throughput [53]. This has motivated extensive research towards `int8` quantization and inference, made possible by the intrinsic error tolerance of neural networks [127, 128].

Recently, several works have also explored algorithmic solutions to reduce the number of operations and the memory footprint, exploiting sparsity [96], adopting smaller convolutional kernel sizes [13], channel shuffling [129], using group convolutions [13, 30], and depthwise separable convolutions [16]. However, dense, compute-heavy, 3×3 convolutional layers are still highly present in many state-of-the-art computer vision models [130–132].

The Winograd convolution algorithm [51] is an interesting optimization opportunity. It applies transformations to the input feature maps and weights using constant transformation matrices. It lowers the 3×3 convolution into an element-wise matrix multiplication with fewer general multiplications. Specifically, the Winograd algorithm works on sub-tiles of the feature maps of dimension $m \times m$, and the larger the tiles ($m \uparrow$), the higher the operations count reduction (MACs $\downarrow$). However, increasing m also causes higher sensitivity to numerical inaccuracies [133], decreasing the overall accuracy of the networks. As a result, $m \in \{2,4\}$ has received the majority of attention in actual implementations, potentially reducing the number of MACs by $2.25\times$ for $m = 2$ and by $4\times$ for $m = 4$. Unfortunately, a simple adoption of `int8` operations is impossible for Winograd with $m = 4$ due to the resulting numerical instability [52, 134, 135]. Moreover, the Winograd algorithm increases the heterogeneity of the compute operations that need to be mapped on the chosen hardware accelerator, and some of these operations, i.e., the transformations, can not be efficiently computed by the large 2D or 3D matrix-matrix multiplication engines that are the cores of modern architectures. Our research aims to enable `int8` Winograd with $m = 4$ on domain-specific accelerators, with a new quantization algorithm that falls in the class of QAT methodologies. This algorithm is combined with architectural and micro-architectural design space exploration to facilitate efficient hardware deployment.

This chapter will provide the background necessary to understand the Winograd algorithm and how it can be used for convolutions, then a brief review of the related works on the Winograd algorithm, quantization, and hardware accelerators (Section 2.2). Next, it presents the proposed tap-wise quantization algorithm (Section 2.3) and techniques for hardware acceleration (Section 2.4). Finally, the algorithm and the system are extensively evaluated for several workloads (Section 2.5).

2.2 Background and Related Works

2.2.1 Winograd Minimal Filtering Algorithm

To compute m outputs, a discrete $(r-1)$-order finite input response (FIR) filter requires $m \cdot r$ multiplications, as shown in Fig. 2.1. In [51], S. Winograd demonstrates the possibility of a minimal filtering algorithm, hereon denoted and referred to as

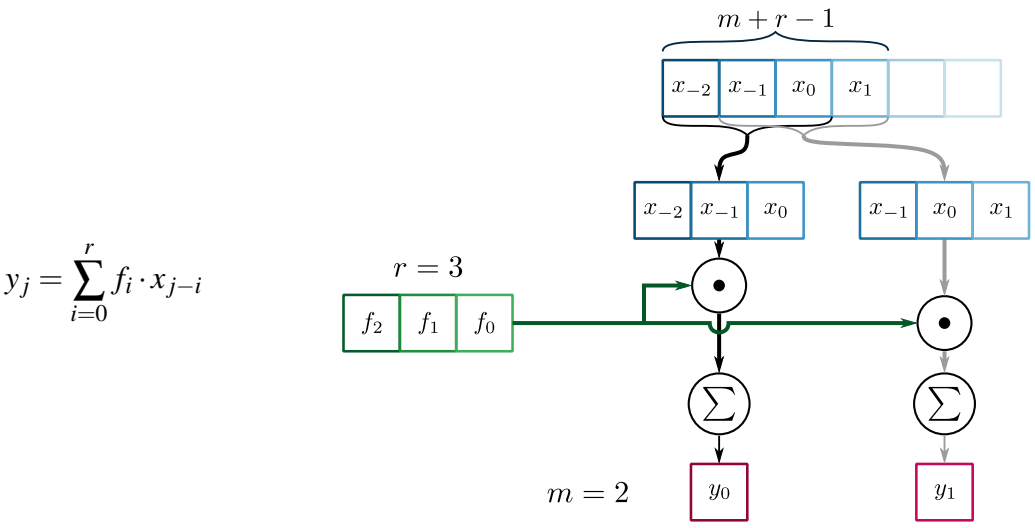


Fig. 2.1 FIR filter

Winograd $F_{m,r}$, requiring only $m+r-1$ multiplications to compute m outputs, i.e., one multiplication per input.

For the case of $F_{2,3}$, the algorithm documented in [51] is the following:

$$\begin{bmatrix} y_0 & y_1 \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 \\ x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} z_1 + z_2 + z_3 \\ z_2 - z_3 - z_4 \end{bmatrix} \tag{2.1}$$

with

$$\begin{aligned} z_1 &= (x_0 - x_2) f_0 & z_2 &= (x_1 + x_2) \frac{f_0 + f_1 + f_2}{2} \\ z_4 &= (x_1 - x_3) f_2 & z_3 &= (x_2 - x_1) \frac{f_0 - f_1 + f_2}{2} \end{aligned} \tag{2.2}$$

From Eq. (2.2), the number of general multiplications needed is 4, as expected from $m+r-1 = 2+3-1 = 4$. However, the algorithm introduces 4 additions on the input data x, 3 additions and 2 multiplications by a constant on the filter coefficients f, and 4 additions to compute the output y. Nevertheless, additions and shifts are much faster and less power-consuming than general multiplications [53].

The same algorithm can be rewritten in a more convenient matrix form as:

$$y = A^T[(Gf) \odot (B^T x)] \tag{2.3}$$

where $\odot$ is the symbol for Hadamard product, or element-wise multiplication, and $B^T \in \mathbb{R}^{(m+r-1)\times(m+r-1)}$, $G \in \mathbb{R}^{(m+r-1)\times(r)}$, and $A^T \in \mathbb{R}^{(m)\times(m+r-1)}$ are constant *transformation matrices*. In particular, for $F_{2,3}$, the matrices assume the values:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{2.4}$$

For Winograd $F_{4,3}$, the number of general multiplications needed is 6, compared to the 12 of the standard algorithm, and the transformation matrices are:

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \quad G = \frac{1}{3} \begin{bmatrix} 3/4 & 0 & 0 \\ -1/2 & -1/2 & -1/2 \\ -1/2 & 1/2 & -1/2 \\ 1/8 & 1/4 & 1/2 \\ 1/8 & -1/4 & 1/2 \\ 0 & 0 & 27 \end{bmatrix} \tag{2.5}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

Inspecting Eq. (2.4) and Eq. (2.5), it can be seen that the transformation matrices for $F_{2,3}$ are relatively sparse, and the coefficients only require additions, subtractions, and a division by 2, which is easily implemented in hardware with a right shift. On the contrary, the $F_{4,3}$ coefficients cover a broader numerical range and require a higher computational effort. As demonstrated in [51], the complexity of the transformation matrices grows for increasing m and r. Therefore, despite reducing the general multiplications, the Winograd algorithm with a high m or r is not practically deployable due to numerical instability problems. Thus, the focus of actual imple-

mentations has been mainly put towards $m \in \{2,4,6\}$, while r depends on the filter characteristics and therefore determines the Winograd algorithm applicability.

2.2.2 Winograd for Convolution

The Winograd 1D algorithm can be nested to obtain higher-dimensional filters. For 2D filtering, Eq. (2.3) becomes:

$$y = A^T[(GfG^T) \odot (B^T xB)]A \tag{2.6}$$

The input x, filter g and output y will be square tiles, with $x \in \mathbb{R}^{(m+r-1)\times(m+r-1)}$, $g \in \mathbb{R}^{(r)\times(r)}$, and $y \in \mathbb{R}^{(m)\times(m)}$. While for the standard 2D filtering algorithm, the number of required general multiplications is $(m \cdot r)^2$, for 2D Winograd filtering, it goes down to $(m+r-1)^2$. Fig. 2.2 shows the arithmetic complexity reduction for varying m and $r = 3$. In particular, the reduction is $2.25\times$ for $F_{2,3}$, and $4\times$ for $F_{4,3}$.

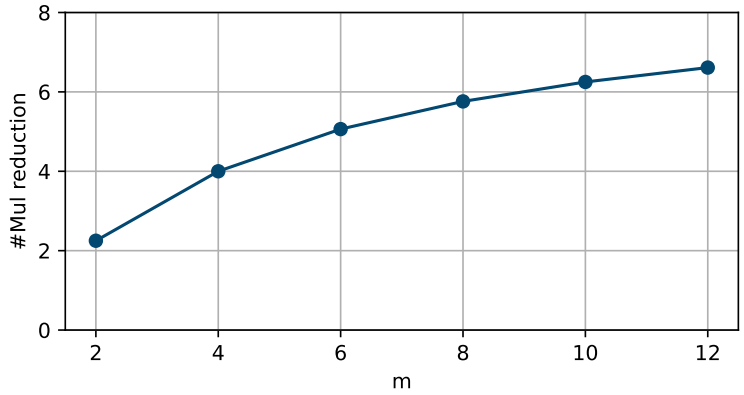


Fig. 2.2 General multiplications reduction of 2D Winograd $F_{m,r}$ w.r.t. the standard filtering algorithm.

2D Winograd algorithm can be applied to convolutional layers with kernels of size $r \times r$. Most Conv layers in SOTA CNNs have 3×3 filters. Therefore, we will focus on Winograd $F_{m,3}$, for simplicity denoted as F_m hereon. Let's consider a single-input-channel-single-output-channel convolution, with input feature map iFM$\in \mathbb{R}^{H\times W}$ and kernel $w \in \mathbb{R}^{r\times r}$. The iFM is divided in spatial tiles of size $(m+r-1) \times (m+r-1)$, with a stride between the tiles of m, yielding to $\lceil \frac{H}{m} \rceil \lceil \frac{W}{m} \rceil$ tiles in total. Each tile $x_{\tilde{h},\tilde{w}}$, where $\tilde{h}$ and $\tilde{w}$ are tile coordinates, is transformed to the Winograd domain by applying $B^T xB$, and the same for the kernel with GwG^T.

Once in the Winograd domain, each transformed input tile $V \in \mathbb{R}^{(m+r-1)\times(m+r-1)}$ is element-wisely multiplied by the transformed weight tile $U \in \mathbb{R}^{(m+r-1)\times(m+r-1)}$ and the resulting tiles are back-transformed to the spatial domain with $A^T YA$. The algorithm is described in Eq. (2.7) and Fig. 2.3.

$$\begin{aligned} y_{\tilde{h},\tilde{w}} &= f * x_{\tilde{h},\tilde{w}} \\ &= A^T[(GfG^T) \odot (B^T x_{\tilde{h},\tilde{w}} B)]A \\ &= A^T[U \odot V_{\tilde{h},\tilde{w}}]A \end{aligned} \tag{2.7}$$

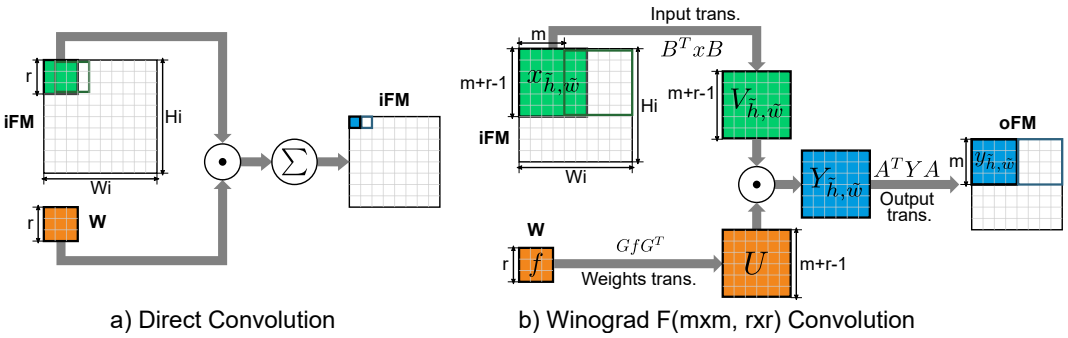


Fig. 2.3 Comparison between standard convolution and Winograd convolution for the single-input-single-output channel case

In the case of multiple input and output channels, the same operation described above needs to be replicated for all the tiles along the input and output channels (Eq. (2.9)). To amortize the cost of the back-transformation $A^T YA$, the accumulation along the input channel dimension can be moved into the Winograd domain (Eq. (2.10)). Eq. (2.8) and Eq. (2.11) are depicted graphically in Fig. 2.4.

$$y_{c_o,\tilde{h},\tilde{w}} = \sum_{c_i=0}^{C_i} f_{c_o,c_i} * x_{ci,\tilde{h},\tilde{w}} \tag{2.8}$$

$$= \sum_{c_i=0}^{C_i} A^T[(Gf_{c_o,c_i}G^T) \odot (B^T x_{ci,\tilde{h},\tilde{w}} B)]A \tag{2.9}$$

$$= A^T[\sum_{c_i=0}^{C_i} (Gf_{c_o,c_i}G^T) \odot (B^T x_{ci,\tilde{h},\tilde{w}} B)]A \tag{2.10}$$

$$= A^T[\sum_{c_i=0}^{C_i} U_{c_o,c_i} \odot V_{ci,\tilde{h},\tilde{w}}]A \tag{2.11}$$

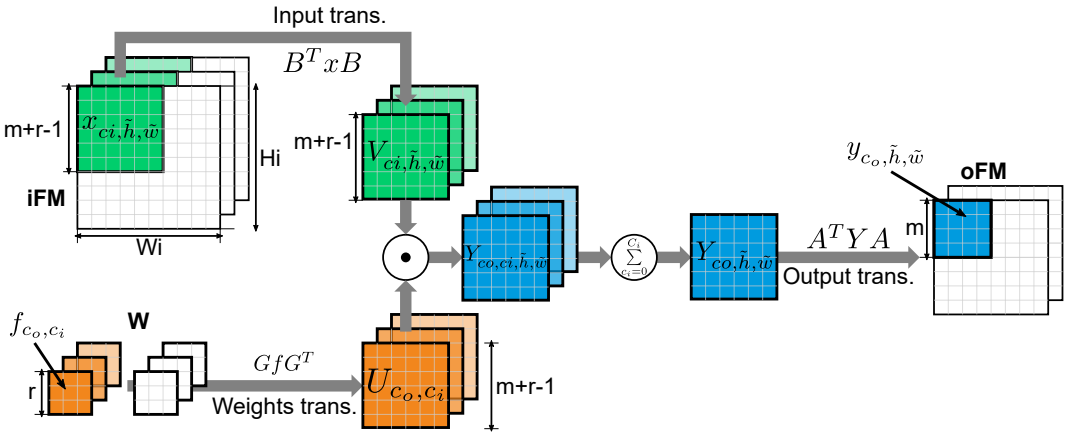


Fig. 2.4 Winograd convolution for convolution with multiple input and output channels.

If we examine only the Hadamard product in the square brackets of Eq. (2.11) and consider one single element of the tiles, hereon denominated *tap* and identified with the indexes (χ, ν), we obtain:

$$M^{(\chi,\nu)}_{c_o,\tilde{h},\tilde{w}} = \sum_{c_i=0}^{C_i} U^{(\chi,\nu)}_{c_o,c_i} \cdot V^{(\chi,\nu)}_{ci,\tilde{h},\tilde{w}} \tag{2.12}$$

By flattening the two dimensions $\tilde{h}$ and $\tilde{w}$ in a single dimension $\tilde{hw}$, Eq. (2.12) can be seen as a matrix multiplication. In conclusion, once in the Winograd domain, by working on the different taps independently, the convolution can be mapped to a series of MatMuls, which, as seen in Chapter 1, are efficiently implemented in hardware. The mapping of the Winograd Convolution to MatMuls is depicted in Fig. 2.5.

2.2.3 Related Works

Winograd Algorithm. The original Winograd algorithm has been extended to work on general 2D convolution in several works [52, 136, 137]. Its performance has been improved by combining it with the Strassen algorithm [138] or by increasing its numerical accuracy by using higher-order polynomials [135] and better polynomial root points for $m > 4$ [133, 139]. Li et al. [140] merged the Winograd method with AdderNet, which substitutes additions for all MAC operations by extracting features using the ℓ_1 norm rather than the ℓ_2 norm. The proposed strategy, however, reduces accuracy from 92.3% for the FP32 baseline to 91.6% for CIFAR-10/ResNet-20.

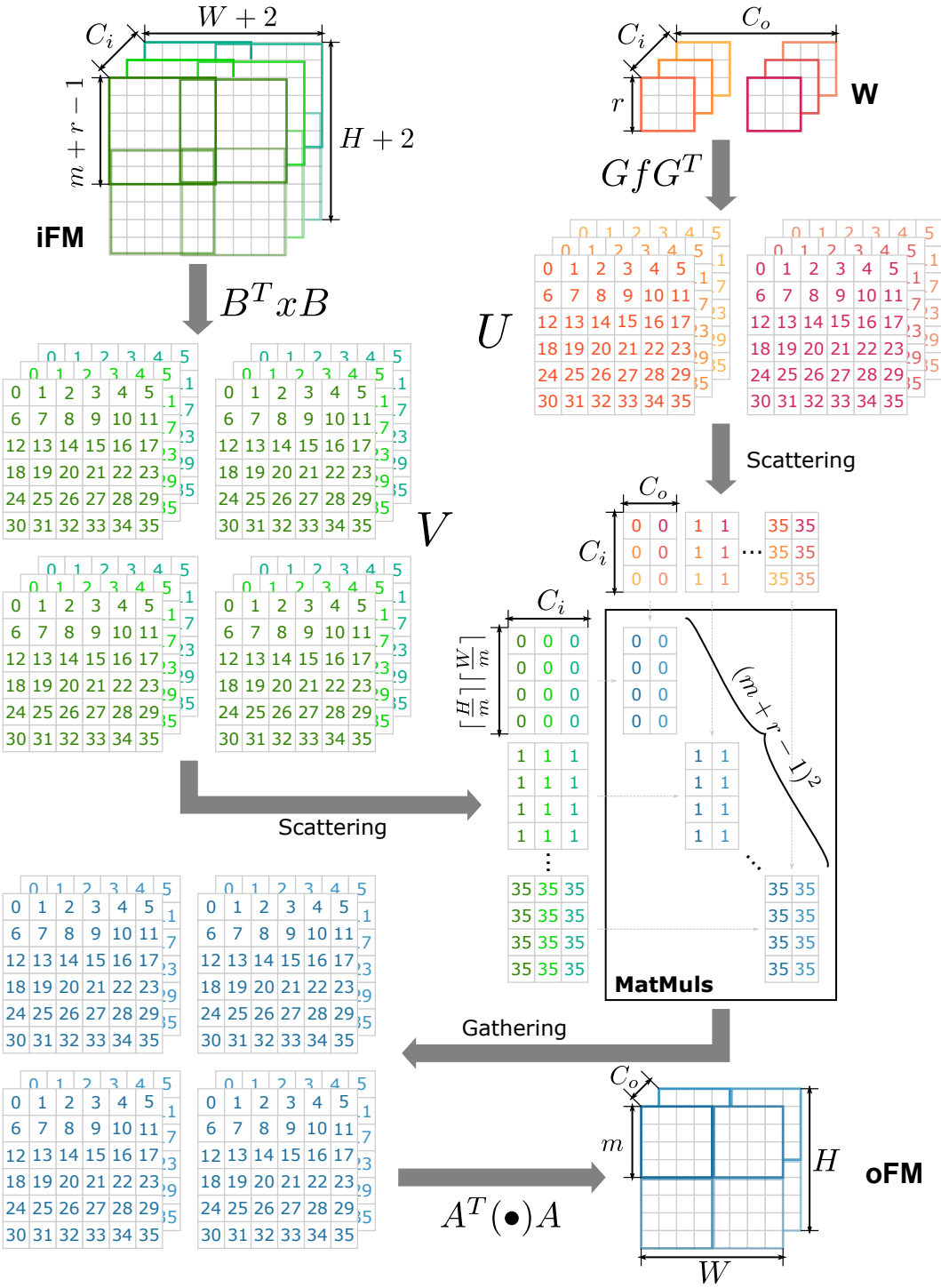


Fig. 2.5 Winograd Convolution mapping to MatMuls.

Sparsity has played a significant role in lowering CNN computational complexity. Liu et al. [141], and Li et al. [142] suggested moving the ReLU activation after the input transformation and pruning the weights once in the Winograd domain. However, they only discuss the drop in MACs on FP32 networks. An intriguing area for further research is the combination of pruning with tap-wise quantization and evaluating its performance on hardware accelerators.

Quantized Winograd. Gong et al. [143] and Li et al. [144] proposed to quantize F_2 in the Winograd domain with a single scaling factor per transformation. The algorithm was extended by Meng et al. [145] using complex root points and increasing the number of valid root points for Winograd F_4. Liu et al. [146] proposed to integrate Winograd and Residue Number System (RNS), using 8 bit moduli 251,241,239 and Winograd F_{14}. Fernandez et al. [134] proposed Winograd-aware training for Quantized Neural Networks with gradients propagated through the Winograd Domain. In the case of F_4, retraining of the transformation matrices (WA-flex) was needed, making the transformation operation dense and introducing FP32 MACs. Barabasz et al. [135] extended the work of Fernandez et al. [134] with Legendre polynomial bases, requiring six additional sparse diagonal matrix multiplications. LoWino [147] proposed to use FP32 feature maps and weights but quantizing them in the Winograd domain. This way, the elementwise multiplication can be computed in `int8`, while input and output transformations are performed in FP32.

Custom Winograd Accelerators. To speed up the Winograd algorithm, several specialized FPGA-targeted accelerators were presented [148–150]. While they have a spatial architecture that can only execute the Winograd algorithm, we provide a method to incorporate Winograd support into a programmable AI accelerator based on a high-throughput MatMul unit, the most popular ASIC accelerator architecture. Wang et al. [151] suggested a RISC-V extension to handle Winograd transformations effectively. The F_2 Winograd operator was proposed to be mapped using Vector Units on a general-purpose edge device by Xygkis et al [152]. WinDConv [153], an accelerator based on NVDLA [154] that supports the F_2 Winograd operator with a fused datapath, is the solution that comes the closest to our research. A direct comparison is unfortunately impossible because they focused on a mobile application situation, provided a post-synthesis-only evaluation utilizing a much more recent technical node, and ignored the impact of external memory on performance. In contrast, our solution produces an average gain in the energy efficiency of 2.1× for 12 SOTA CNNs, whereas their Winograd extension increases energy efficiency over

their baseline of 1.82× in the optimal case, i.e., with 100% utilization. They also quantize to 6 bits in the spatial domain, resulting in a greater accuracy loss [155] than the tap-wise quantization flow we propose.

Winograd SW optimizations. For GPUs and CPUs, several effective SW implementations of the Winograd algorithms have recently been developed [156–160]. These works use similar loop-level optimization techniques, like loop unrolling, parallelization, or vectorization, to maximize the performance of the targeted platforms, which have very different limitations and features than ours.

2.3 Tapwise Quantization

As described in Chapter 1, moving from a floating-point to a fixed-point representation improves hardware and computation efficiency in terms of area occupation, energy, and throughput.

To guarantee a lossless computation in the Winograd domain, it is necessary to analyze the number of bits required to represent the values after the transformation. Eq. (2.13) shows the unrolled $B^T xB$ operation for Winograd F_2.

$$B^T xB = \begin{bmatrix} x_{0,0}-x_{0,2}-x_{2,0}+x_{2,2} & x_{0,1}+x_{0,2}-x_{2,1}-x_{2,2} & -x_{0,1}+x_{0,2}+x_{2,1}-x_{2,2} & x_{0,1}-x_{0,3}-x_{2,1}+x_{2,3} \\ x_{1,0}-x_{1,2}+x_{2,0}-x_{2,2} & x_{1,1}+x_{1,2}+x_{2,1}+x_{2,2} & -x_{1,1}+x_{1,2}-x_{2,1}+x_{2,2} & x_{1,1}-x_{1,3}+x_{2,1}-x_{2,3} \\ -x_{1,0}+x_{1,2}+x_{2,0}-x_{2,2} & -x_{1,1}-x_{1,2}+x_{2,1}+x_{2,2} & x_{1,1}-x_{1,2}-x_{2,1}+x_{2,2} & -x_{1,1}+x_{1,3}+x_{2,1}-x_{2,3} \\ x_{1,0}-x_{1,2}-x_{3,0}+x_{3,2} & x_{1,1}+x_{1,2}-x_{3,1}-x_{3,2} & -x_{1,1}+x_{1,2}+x_{3,1}-x_{3,2} & x_{1,1}-x_{1,3}-x_{3,1}+x_{3,3} \end{bmatrix} \tag{2.13}$$

In the worst case, $B^T xB$ requires only 2 extra bits, as the sum of k n-bit integer values needs a $\lceil \log_2(k(2^n-1)+1) \rceil$-bit integer to represent the result. By doing a similar analysis, GfG^T for F_2 requires 3 extra bits. Nevertheless, because the weight and activation value distributions in CNNs typically follow a Gaussian distribution centered around zero, 8 bits are sufficient not to deteriorate the accuracy.

For Winograd F_4, $B^T xB$ and GfG^T transformations require 8 and 10 extra bits respectively for a bit-true computation. This increased bitwidth would lead to a significant rise in power and area costs in a hardware accelerator. At the same time, quantizing the transformed tiles to `int8` in a traditional fashion, i.e., using the same scaling factor for all the taps, unacceptably degrades the accuracy of the network [134].

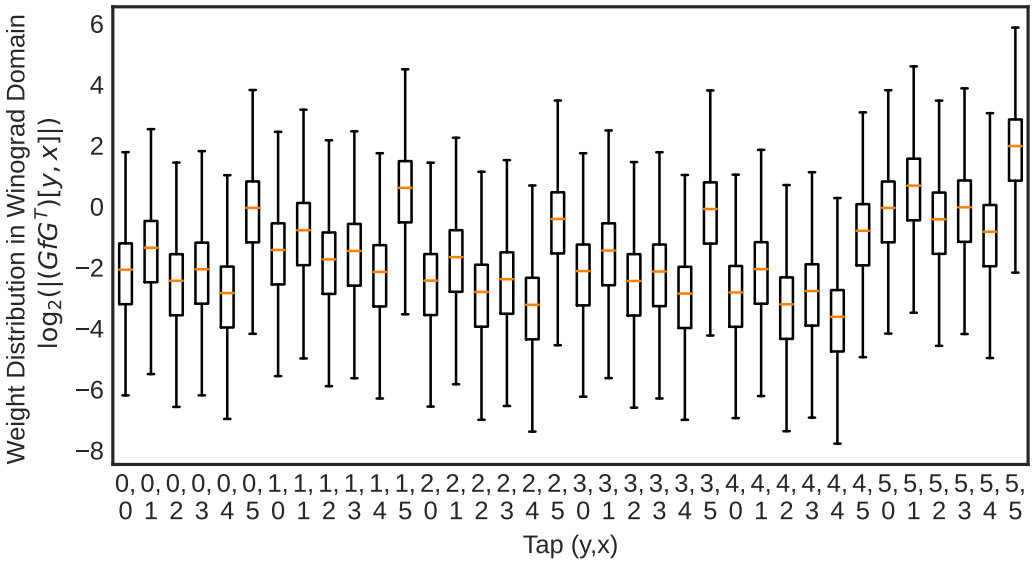


Fig. 2.6 Weight Distribution in Winograd domain of GfG^T taps for ResNet-34 on ImageNet

To this end, a *tap-wise quantization algorithm* is proposed, enabling `int8` inference on Winograd F_4 convolutional layers. Winograd transformations change the dynamic range of the outputs, resulting in the necessity of a higher number of bits to represent the values accurately. However, inspecting the numerical distribution post-transformations, we see that the distribution heavily depends on the tap index. An example of this behavior is provided in Fig. 2.6, where the value distribution of each GfG tap is shown. Each tap needs ~8 bits to cover its dynamic range, while ~15 bits are required for the combined distribution of all the taps. Given these analyses, we propose to quantize each tap independently with a different scaling factor.

With quantization, floating-point numbers are approximated as integers numbers:

$$x_{\texttt{int}n} = \textbf{clamp}\left(\textbf{round}\left(\frac{x}{s}\right), -2^{n-1}, 2^{n-1}-1\right) \tag{2.14}$$

where $s = \frac{x_{max}}{2^{n-1}}$ is a scaling factor and x_{max} is the largest representable value. Considering the 2D Winograd convolution equation (Eq. (2.9)), and applying standard quantization, i.e., using a uniform scaling factor for all the values, we obtain:

$$y = \sum_{c_i=0}^{C_i} A^T \left[\sigma_G \left\lceil \left(Gf_{c_i}G^T\right)/\sigma_G \right\rceil_{\texttt{int}n} \odot \sigma_B \left\lceil \left(B^T x_{ci} B\right)/\sigma_B \right\rceil_{\texttt{int}n}\right] A \tag{2.15}$$

where σ_G and σ_B are scalar scaling factors for the transformed weights and iFMs respectively. For a lighter notation, indexes c_o, $\tilde{h}$, and $\tilde{w}$ have been omitted w.r.t. Eq. (2.9).

To quantize each tap independently, i.e., with a different scaling factor for each tap, σ_G and σ_B need to be replaced with tap-wise scaling matrices $\Sigma_G, \Sigma_B \in \mathbb{R}^{(m+r-1)\times(m+r-1)}$ (Eq. (2.16)). By applying the distributivity law and rearranging the linear operations, we obtain Eq. (2.17). After input and weight transformations, the tiles are scaled and quantized to calculate the multiplications and accumulations in the integer domain. Then, rescaling by $\Sigma_{GB} = \Sigma_G \odot \Sigma_B$ is performed only once before transforming the tiles back into the spatial domain. In Eq. (2.16), Eq. (2.17), and Eq. (2.18), the symbol $\oslash$ represents the inverse operation of the Hadamard product, i.e., element-wise division.

$$y =$$

$$= A^T \left[\sum_{c_i=0}^{C_i} \Sigma_G \odot \left\lceil \left(G f_{c_i} G^T\right) \oslash \Sigma_G \right\rfloor_{\texttt{int}b} \odot \Sigma_B \odot \left\lceil \left(B^T x_{ci} B\right) \oslash \Sigma_B \right\rfloor_{\texttt{int}b} \right] A \quad (2.16)$$

$$= A^T \left[\Sigma_G \odot \Sigma_B \odot \sum_{c_i=0}^{C_i} \left\lceil \left(G f_{c_i} G^T\right) \oslash \Sigma_G \right\rfloor_{\texttt{int}b} \odot \left\lceil \left(B^T x_{ci} B\right) \oslash \Sigma_B \right\rfloor_{\texttt{int}b} \right] A \quad (2.17)$$

$$= A^T \left[\Sigma_{GB} \odot \sum_{c_i=0}^{C_i} \left\lceil \left(G f_{c_i} G^T\right) \oslash \Sigma_G \right\rfloor_{\texttt{int}b} \odot \left\lceil \left(B^T x_{ci} B\right) \oslash \Sigma_B \right\rfloor_{\texttt{int}b} \right] A \quad (2.18)$$

2.3.1 Winograd-Aware Training

The tested networks are trained with stochastic gradient descent. Standard convolutional layers could be replaced with Winograd layers post-training, as it is simply a different implementation of the same operations. However, to take into account during training the slight numerical fluctuations introduced by Winograd transformations and, therefore, to improve the training accuracy when Winograd layers are used, we also adopt the static Winograd-aware training method [134], propagating the gradients through the Winograd domain:

$$\left[\frac{\partial L}{\partial f} = \frac{\partial (A^T [(GfG^T) \odot (B^T x B)] A)}{\partial f} \frac{\partial L}{\partial Y} \right] \quad (2.19)$$

Since using a filter size with $r > 3$ leads to the same numerical problems discussed for increasing m, we only implement as Winograd layers those with kernel size 3×3 and unitary stride. Although the Winograd algorithm can be used to produce strided convolution [136, 137], the control and computation complexity predominate the potential MACs reduction (i.e., stride-2 F_4 leads only to a $1.8\times$ MACs reduction w.r.t. the $4\times$ reduction of stride-1 F_4).

2.3.2 Power-of-Two Tapwise Quantization

The scaling by Σ_G, Σ_B, and Σ_{GB} introduce one floating-point multiplication per transformation and tap, increasing the computational cost of the Winograd algorithm. Thanks to data reuse strategies, the overall cost of the transformations and scaling can be amortized, e.g., reusing the transformed iFMs for multiple oFMs. However, limiting the scaling matrices to power-of-two values is advantageous, allowing both the transformations and the rescaling to be carried out using simply shift-and-add operations. We examine and combine three different methodologies to determine the power-of-two scaling factors.

1. **Straight-forward power-of-two quantization.** The scaling factors, derived from the calibrated maximum values, are rounded to the next power-of-two:

$$\tilde{s}_{i,j} := 2^{\lceil \log_2 s_{i,j} \rceil} \tag{2.20}$$

   In this way, the quantized value becomes:

$$q_{\texttt{int}b}(x) := \left\lfloor x / 2^{\lceil \log_2 s \rceil} \right\rceil_{\texttt{int}b} \tag{2.21}$$

2. **Learned power-of-two quantization.** The scaling factors are learned during training to find a better representation range, as, for example, enhancing the precision of smaller values could be more crucial for the end-to-end accuracy than having less clamped values. Since the quantization function is a step function with derivative zero almost everywhere, thus preventing the gradient backpropagation, *straight-through estimator* is applied [161]:

$$\frac{\partial}{\partial x}\lceil x \rceil = \frac{\partial}{\partial x}\lceil x \rfloor = \frac{\partial}{\partial x}\lfloor x \rfloor = 1 \tag{2.22}$$

To better adapt to the power-of-two quantization, rather than training the scale value s, we compute the gradient to the logarithm of $\log_2 t$, s.t. $s = 2^{\lceil \log_2 t \rceil}$ [162]:

$$\frac{\partial q(x)}{\partial \log_2(t)} = s \ln(2) \cdot \text{clamp}\left(\left\lfloor \frac{x}{s} \right\rceil - \frac{x}{s}, -2^{b-1}, 2^{b-1} - 1 \right) \tag{2.23}$$

We apply the Adam optimizer to the scale factors for faster convergence and scale invariance, with its built-in gradient normalization ($\beta_1 = 0.9$, $\beta_2 = 0.99$) [163]. As for the other parameters, stochastic gradient descent with an independent learning rate is used.

3. **Knowledge distillation.** KD consists in training a compact network (*student*) by minimizing its distance to a larger network (*teacher*). In the proposed training flow, we use the floating-point baseline model as the teacher network and the power-of-two tap-wise quantized network as the student. We adopt the Kullback-Leibler divergence loss and the tempered softmax activation function [107].

2.4 Hardware Acceleration

Since a high-area and high-throughput datapath for MatMuls is the core component of almost all modern accelerators [164, 69, 165], it is challenging to convert the Winograd algorithm's lower computational complexity into a wall-clock time speed-up. As shown in Section 2.2.2, of all the steps of the Winograd algorithm, only the element-wise multiplications can be effectively mapped to batched MatMuls. On the other hand, a 2D or 3D MatMul engine can not process at high throughput the input, output, and weight transformations, as they involve several tiny MatMuls and data-layout rearrangements. Thus, executing a convolutional layer with the Winograd algorithm rather than the standard approach, i.e., im2col, moves "ops" from cheap, high-arithmetic intensity operations to more sparse, low-arithmetic intensity ones.

Moreover, the Winograd method increases the heterogeneity of the compute operations by introducing a new class of operations, making the coordination of data movements and computations and the balancing of memory bandwidth and computing significantly more difficult. Additionally, compared to the standard implementation, the Winograd algorithm reduces the computational complexity of

the Conv2D operation, but at the expense of significantly lowering the potential for data reuse. Thus, the pressure on memory bandwidth is unavoidably increased, calling for a careful dataflow construction.

To address these challenges, we provide a design space exploration of area- and power-efficient custom hardwired modules to implement the low-arithmetic intensity "ops" of the Winograd transformation operations. Next, we demonstrate how to integrate Winograd transformation engines in an industrial-grade, programmable AI accelerator and how to adjust the microarchitecture of such blocks to match the throughput of data movement, Winograd transformation, and compute operations, thereby maximizing overall compute efficiency.

2.4.1 Baseline Accelerator

The architecture of our baseline inference DSA, which includes two DaVinci-inspired AI cores (AICs) [69] AIC0 and AIC1, is depicted in Fig. 2.7. Each core provides the functionality required for processing CNN layers and exposes a custom instruction set architecture (ISA). The datapath of the AI core consists of a *Cube Unit* for MatMuls, and a *Vector Unit* and *Scalar Unit* for scalar and vector operations, respectively.

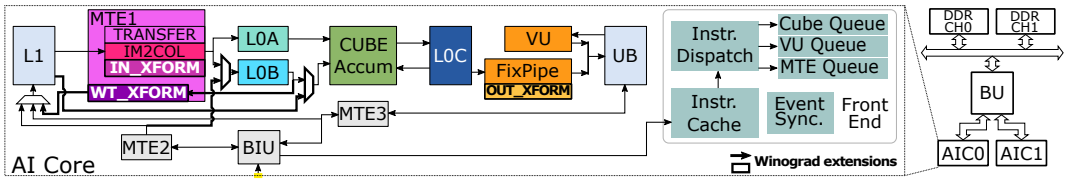


Fig. 2.7 High-level overview of the inference accelerator with the proposed extensions

The Cube Unit (Fig. 2.8) can work in `int` and `float` mode. In `int` mode, it performs a MatMul between two `int8` matrices of size $[16 \times 32]$ per cycle, producing an `int32` $[16 \times 16]$ output matrix which can be accumulated to a third input operand. In `float` mode, the inputs are `float16` $[16 \times 16]$ matrices, and the output is a `float32` $[16 \times 16]$ matrix.

The data layout adopted for the feature maps, i.e., the *fractal layout* [166], is such that it simplifies the memory access patterns. The reduction axis of the feature maps (C) is split into a sub-dimension C_0 and a super-dimension $C_1 = \lceil \frac{C}{C_0} \rceil$, where $C_0 = 32$ for `int` mode, and $C_0 = 16$ for `float` mode. The elements along the sub-dimension

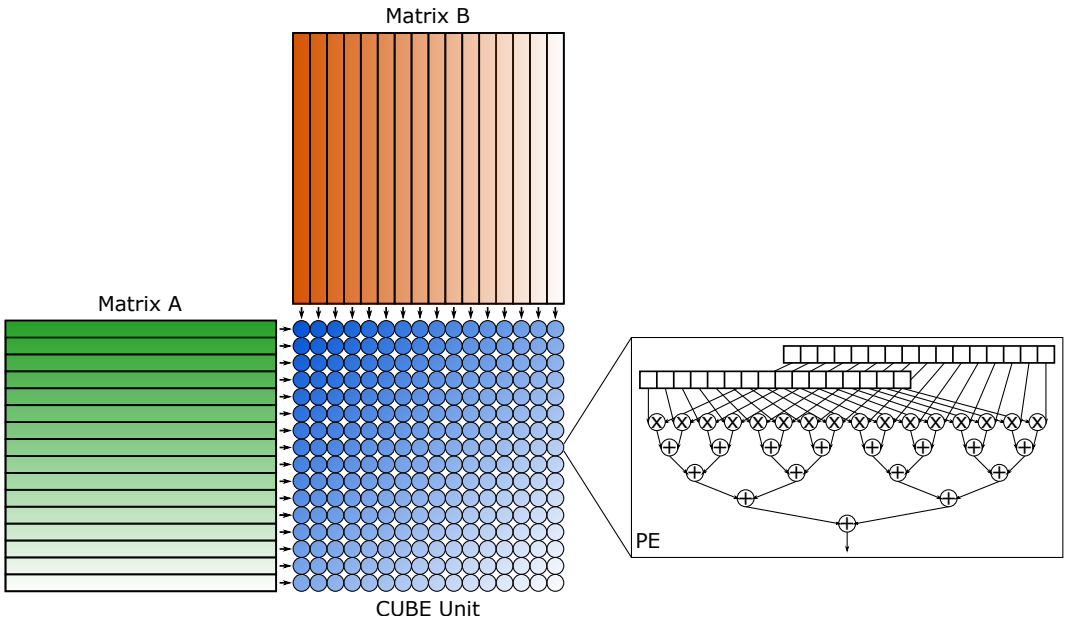


Fig. 2.8 Cube Unit.

C_0 are stored contiguously in memory. With this format, the layout of a $\langle N,C,H,W\rangle$ iFM becomes $\langle N,C_1,H,W,C_0\rangle$. Fig. 2.9 graphically shows the fractal layout.

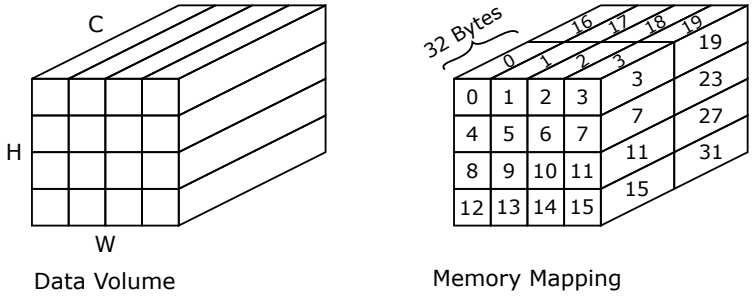


Fig. 2.9 Fractal data layout.

The Vector Unit is 256B wide, with a peak throughput of 256 `int8` and 128 `float16` ops per cycle. It supports general arithmetic operations between vectors and specialized operations often appearing in CNNs, such as ReLU or pairwise reductions. It is also the unit responsible for data type conversions and data layout transformations. For relevant CNNs workloads, the width of the Vector Unit assures that its throughput matches the output data rate of the Cube Unit.

The on-chip memory is organized in a multi-level hierarchy. *L1* is the second level memory, *L0C* serves as input and output buffer to the *Vector Unit*, L0A and L0B as input buffers for the *Cube Unit*, and *L0C* as its output. Data movements and

layout transformations throughout all the memory hierarchy levels are fully software managed by the on-chip memory transfer engines (MTEs). In particular, the *MTE2* is responsible for the data transfer from the global memory (*GM*) to either the unified buffer *UB* or *L1*, while vice-versa the *MTE3* moves data from the *UB* to *GM* or *L1*. The *MTE1* moves data from *L1* to *L0A* and *L0B*, and, in case of Conv operation, is responsible for performing *im2col* transformation on the tiles to lower the operation to a MatMul. The *im2col* engine within the *MTE1* supports kernel sizes 3, 5, and 7, and strides 1 and 2. The possibility of transferring data from *UB* to *L1* is needed in case of multiple layers fusion, i.e., multiple layers can be computed without ever needing to transfer data off-chip. For the same purpose, a *FixPipe* module within the *Vector Unit* transfers the output of the *Cube Unit* from *L0C* to *UB*, performing re-quantization on the fly if needed.

The size and number of banks of the on-chip memories are tailored to minimize area while maintaining sufficient bandwidth and capacity to prevent computational units from being blocked. In particular, *L0A* and *L0B* can both feed one operand per cycle to the *Cube Unit*, i.e., a tile of 256B each, without any bank conflict. In the same way, *L0C* sustains the rate of one read and write operation per cycle from the *Cube Unit*, and it also has a read port towards the *FixPipe* module. *L1* manages bank conflicts at runtime through a complicated addressing system and multiple read and write ports. Adopting data reuse in other memories allows the idle cycles introduced by the bank conflicts to be excluded from the computational critical path.

The AI core uses an in-order scalar front-end to offload the instructions to the different units. Each unit has a separate instruction queue, allowing the units to run in parallel. To reduce the instruction dispatching overhead, the ISA of the core allows setting a repetition factor for each instruction, together with the stride for all the operators. The repetitions are then handled by each unit internally through a μ-sequencer. To manage data dependencies, the different units are synchronized explicitly with an explicit token exchange mechanism [167], with a form of *decoupled access/execute strategy* [168] that allows the programmer to handle the overlap of compute operations and data movements.

2.4.2 Winograd Transformation Engines

The input, weights, and outputs Winograd transformations $B^T xB$, GfG^T, and $A^T YA$ of Eq. (2.6) can be generalized as follows:

$$s_w = T^T \times s \times T = T^T \times \tilde{s}, \tag{2.24}$$

where T is a generic constant transformation matrix of size $[h_T \times w_T]$, s is a tile to be transformed with shape $[h_T \times h_T]$. s could be a tile of the iFMs, oFMs, or of the weights, depending on the transformation being performed. To transform all the iFMs, weights, and oFMs volumes, the Winograd transformations in Eq. (2.24) need to be repeated multiple times, specifically:

- $\frac{H}{m} \times \frac{W}{m} \times C_{in/out}$ for the iFMs;
- $C_{in} \times C_{out}$ for the weights;
- $\frac{H}{m} \times \frac{W}{m} \times C_{in/out}$ for the oFMs

To investigate the different area-throughput trade-offs and to design area- and power-efficient Winograd transformation engines, we unroll the entire $T^T \times s \times T$ operation into a flat dataflow graph (DFG). Then, we apply several techniques to optimize the DFG furtherly, mainly the fact that the T matrix is a constant known at design time:

- Taking advantage of the many common terms and symmetries in the transformation matrices, we apply common element subexpression or other algebraic techniques to share computations and reuse results, reducing the number of computational resources or the number of cycles needed.
- Most of the values in the transformation matrices are powers-of-two, which allows us to avoid multipliers in favor of simpler shifters. When, rarely, non-powers-of-two coefficients are present, the multiplication is carried out as a series of shifts and adds, e.g., $c = 5 \cdot a = (a << 2) + a$.
- When possible, the bitwidth of intermediate results is kept to the minimum to reduce the area and power consumption of the allocated computational resources.

Based on this exploration and optimization, we propose two high-level design strategies, which we refer to as *row-by-row* and *tap-by-tap* transformation engines.

The *row-by-row transformation engine* (Fig. 2.10) comes in two versions, *slow* and *fast*, both based on the decomposition of the transformation operation into a series of vector-matrix multiplications $s[y,:] \times T$, mapped on a spatial processing element (PE).

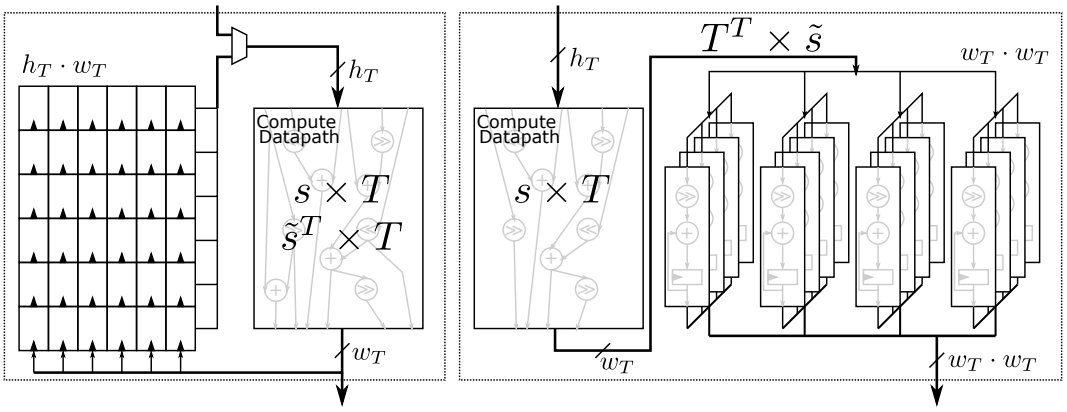


Fig. 2.10 Row-by-row engine in the slow (left) and fast (right) version.

In detail, the PE accesses one row of the matrix s per cycle. Therefore, the whole operation $\tilde{s} = s \times T$ is completed in h_T cycles. Using only shifters and adders, the PE hardcodes the vector-matrix multiplication with matrix T. The second step $T^T \times \tilde{s}$ of the Winograd transformation can be mapped and computed in two ways, either by reusing the resources already instantiated in the PE (*slow solution*) or by allocating additional resources in the PE (*fast solution*).

The slow solution exploits the fact that to map $T^T \times \tilde{s}$ on the same PEs implementing the $\times T$ matrix multiplication it is possible to manipulate the operation to $\tilde{s}^T \times T$, which only requires transposing the intermediate result $\tilde{s}$. For this purpose, an additional set of $h_T \cdot w_T$ registers is needed. This solution produces one row of the output matrix per cycle (w_T cycles needed) after an initial latency of h_T cycles. Overall, the slow solution saves computational resources at the cost of lower throughput.

To avoid lowering the throughput, the fast solution requires adding $w_T \cdot w_T$ lanes to compute s_w in an output-stationary fashion (Fig. 2.10). This solution produces the entire output matrix in a single cycle, after an initial latency of h_T cycles.

As shown in Fig. 2.11, to perform multiple transformations in parallel the PE can be replicated. We denote the two factors controlling the transformations parallelism as P_c and P_s, the former along the channels dimension, and the latter along the spatial dimension. The parallelization strategy is not only constrained by the area budget but also by the memory bandwidth and access pattern requirements. In particular, the row-by-row engine needs enough memory bandwidth to read multiple rows of several tiles along the spatial and channel dimension. Exploiting the overlap between spatially adjacent tiles is possible to alleviate the memory bandwidth pressure. Moreover, the elements of a spatial row need to be stored contiguously in memory.

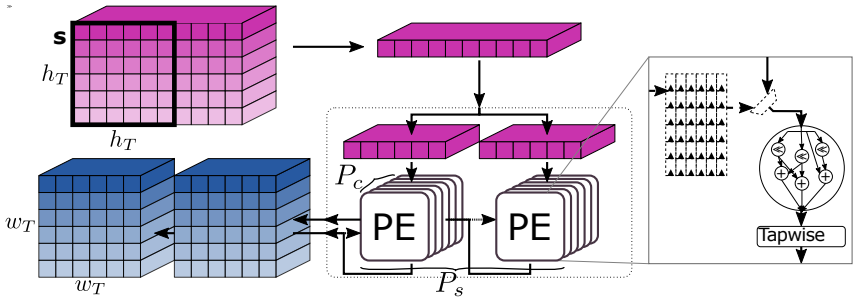


Fig. 2.11 Row-by-row engine with parallelization.

The *tap-by-tap transformation engine* (Fig. 2.12) is located at a different point in the optimization space, with the DFG completely unrolled in time.

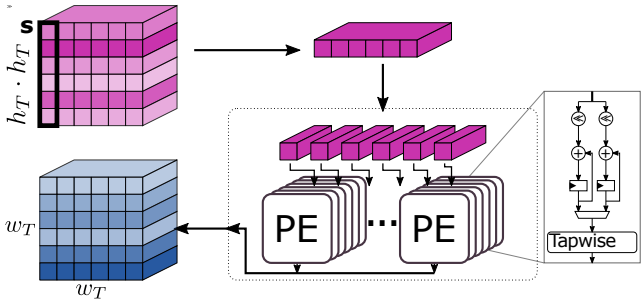


Fig. 2.12 Tap-by-tap engine with parallelization.

In this engine, the PE is very simple and consists only of a configurable shifter to implement the multiplication, an adder/subtractor, and a register serving as an accumulator. Therefore, the PE can implement one MAC per cycle, taking, in the worst case, $h_T \cdot h_T$ cycles to compute a single tap. However, to reduce the total number of cycles, two properties of the transformation matrices can be exploited:

1. The actual number of cycles per tap is usually lower than $h_T \cdot h_T$ cycles, thanks to the sparsity of the transformation matrices.

2. Thanks to the symmetry of the transformation operation, some taps share a sizeable portion of computations with other taps. Therefore, the result of a tap can be reused as an operand for the computation of another tap if this reduces the overall number of MACs. And if the taps computations are scheduled so that the second tap is computed right after the one it shares the operations with, the value to be reused can be simply kept stationary in the accumulator register without the need for additional registers. An example of this technique is provided in Eq. (2.25), taking as an example the operations needed to compute to taps of the GfG^T transformation for F_4.

$$
\begin{aligned}
U_{1,1} &= \frac{1}{4}f_{0,0} + \frac{1}{4}f_{0,1} + \frac{1}{4}f_{0,2} + \frac{1}{4}f_{1,0} + \frac{1}{4}f_{1,1} + \frac{1}{4}f_{1,2} + \frac{1}{4}f_{2,0} + \frac{1}{4}f_{2,1} + \frac{1}{4}f_{2,2} \\
U_{2,1} &= \frac{1}{4}f_{0,0} + \frac{1}{4}f_{0,1} + \frac{1}{4}f_{0,2} - \frac{1}{4}f_{1,0} - \frac{1}{4}f_{1,1} - \frac{1}{4}f_{1,2} + \frac{1}{4}f_{2,0} + \frac{1}{4}f_{2,1} + \frac{1}{4}f_{2,2} \\
U_{2,1} &= U_{1,1} + -\frac{1}{2}f_{1,0} - \frac{1}{2}f_{1,1} - \frac{1}{2}f_{1,2}
\end{aligned}
\tag{2.25}
$$

For the row-by-row engine too, a higher throughput is obtained by allocating multiple PEs and using them to perform transformations in parallel along P_c and P_s. We can also add a parallelization axis P_t, representing the number of taps computed in parallel in a PE. By observing Eq. (2.25), we note that two different taps can be computed from the same input data with different coefficients. Therefore, allocating $P_t = 2$ lanes inside the PE makes it possible to compute two taps in half of the cycles without doubling the necessary read bandwidth. Higher throughput can be achieved by replicating the PEs to perform multiple transformations in parallel. Moreover, P_t does not alter the output bandwidth requirements if the write-back is split into multiple sub-writes.

We add an input or output stage with a programmable shifter and a rounding module to the PE to accommodate tap-wise quantization. The number of parallel taps produced or used during a cycle determines the necessary quantization stages. The overall performance and requirements of the two proposed solutions are reported in Tab. 2.1.

In the following section, we will describe the Winograd Operator's data flow and explain the rationale behind our specific design decisions.

Table 2.1 Performance and bandwidth requirements of the Winograd transformation engines.

	Cycles/Xform [cycles]	**Parallel Xforms**	**RD BW** [B/cycle]	**WR BW** [B/cycle]
Row-by-row				
- *slow*	$h_T + w_T$	$P_c \cdot P_s$	$P_c \cdot P_s \cdot h_T$	$P_c \cdot P_s \cdot h_T$
- *fast*	h_T		$P_c \cdot P_s \cdot w_T \cdot w_T$	
Tap-by-tap	T dependent	$P_c \cdot P_s \cdot P_t$	$P_c \cdot P_s$	$P_c \cdot P_s$

2.4.3 Winograd Operator

The Winograd operator can now be mapped onto the AI core leveraging the dedicated engines to maximize throughput and minimize energy consumption. The proposed dataflow of the Winograd operator for a 3×3 Conv2D layer is reported in Listing 2.1.

```
# WEIGHT Blocking
for wt_tile_gm in WT_GM[2*cout_tile_sz + block_id:...].get_next_tile(
    cin_l0b_tile_sz, cout_l0b_tile_sz,
    double_buffering=True):
  mte2.transfer(wt_tile_gm, wt_tile_l0b)
  mte1.weight_xform(wt_tile_l0b, wt_tile_l1[...])
# IFM L2 Blocking
for ifm_gm_tile in IFM_GM.get_next_tile(
    batch_l2_tile_sz, h_out_l2_tile_sz,
    w_out_l2_tile_sz, double_buffering=True):
  mte2.transfer(ifm_gm_tile, ifm_l2_tile, broadcast=True)
  #IFM L1 Blocking
  for ifm_l1_tile in ifm_l2_tile.get_next_tile(
      batch_l1_tile_sz, h_out_l1_tile_sz,
      w_out_l1_tile_sz, double_buffering=True):
    # IFM L0 Blocking
    for ifm_l0_tile in ifm_l1_tile.get_next_tile(
        bath_l0a_tile_sz, h_out_l0a_tile_sz,
        w_out_l0a_tile_sz, chs_in_l0a_tile_sz,
        double_buffering=True):
      mte1.input_xform(ifm_l0_tile, ifm_l0a_tile)
      cube.mmad(ifm_l0a_tile, wt_tile_l1[...],
                out_l0c_tile[...])
    vec_unit.out_xform(out_l0c_tile, out_prescale_ub_tile)
    vec_unit.dconv(out_prescale_ub_tile, out_postscale_ub_tile, alpha_q)
```

```
26    mte3.write(out_postscale_ub_tile, OUT_GM[:])
```

Listing 2.1 Dataflow of the Winograd operator

First, a tile of weights is moved from *GM* to *L1*, with the transformation to the Winograd domain performed on the fly (lines 2-6). The core parallelism is exploited on the output channels dimension, i.e., each core works on a different subset of output channels (line 2). The data transfer is carried out in tiles (line 5). Each tile is stored in *L0B*, which is used as an intermediate buffer, and transformed to the Winograd domain by the weight transformation engine of the *MTE1*. The resulting tiles are stored in *L1* (line 6). *L0B* is double-buffered to allow the overlap of data transfers from *GM* to *L0B*, and weight transformations from *L0B* to *L1*. Once in *L1*, the weights are kept stationary and reused for all the input feature maps. The synchronization between the transfers handled by the *MT2* and the *MTE1* happens via proper token exchange instructions. For clarity, these synchronization instructions are not reported in the pseudocode.

Second, a tile of the input feature maps is transferred from *GM* to *L1* by the *MTE2* (line 11). The transformation to the Winograd domain is performed by the input transformation engines in the *MTE1*, fetching tiles from *L1* and storing the result in *L0A* (line 21). The element-wise multiplications, mapped to a sequence of MatMuls as explained in Section 2.2, are then executed by the *Cube Unit* (line 22). When the *Cube* finishes the processing of a batch of tiles, the engines within the *Vector Unit* are used to apply the output transformation (line 24). Finally, the *Vector Unit* is also used to apply the final re-quantization (line 25), *MTE3* writes the result back to the *GM* (line 26).

To maximize the concurrency between data movement, compute, and Winograd transformations, double-buffering is applied across the whole on-chip memory hierarchy with three levels of loop blocking:

1. **(lines 8-10)**: the transfer of the iFMs from the *GM* to *L1* (line 11) is done in parallel with all the core-level computations and data movements (lines 13-26).

2. **(lines 16-25)**: the input feature maps transformations and the MatMuls in the *Cube Unit* (lines 17-23) are overlapped with the output transformations, re-quantization and the write-back to *GM* (lines 24-26).

3. **(lines 17-20)**: the input transformations (line 21) and the batched MatMuls (line 22) are overlapped.

Maximizing the *Cube Unit* utilization and thus the compute efficiency requires matching the production and consumption rate of all the units of the AI core. The primary requirement is to match the *Cube Unit* consumption rate with the production rate of the input feature maps transformation engines, as any overhead in the innermost loop would be multiplied by the total number of outer iterations (**lines 17-20**).

As described in Section 2.4.1, the iFMs are stored in *L1* in the fractal data layout $\langle N, C_1, H, W, C_0 \rangle$, and specifically $\langle N, C_1, H, W, 32 \rangle$ when working in `int8` format. Therefore, 32 input channels and the spatial dimension W are contiguous in memory, making the row-by-row engine the most suitable choice as the input transformation engine. Considering the 512B read/write bandwidth of $L1$ and *L0A* respectively, it is possible to replicate the PEs 32 times along the input channel dimension C_{in} and two times along the $W/4$ dimension, performing 64 transformations in parallel. In this way, the *MTE1* input transformation engine has a production rate of $64 \cdot \frac{36}{12} (= 192)$B/cycle, while the *Cube unit* has a consumption rate of 512B/cycle, i.e., the *Cube* is $2.7\times$ faster than the $MTE1$ transformation engines. Consequently, to avoid stalling the *Cube*, at least $3 \times 16 = 48$ output channels must be computed per inner loop (lines 21-22). Since in most networks the number of channels is a power of 2, we build the dataflow and the system in such a way as to be able to compute $4 \times 16 = 64$ output channels per iteration, simplifying the tiling of the loops. This also means that *L0C* has to store at least 4×36 tiles of 16×16 `float32` elements for a total *L0C* size of 288kB considering double-buffering.

It also has to be noted that the row-by-row engine introduces the need to slightly modify the addressing capabilities of *L0A*. To sustain the consumption rate of the *Cube unit*, *L0A* is organized into 16 banks, with a word length of 32B per bank. The addresses of *L0A* need to be 512B-aligned. The row-by-row engine produces multiple taps belonging to the same tile per cycle, and, with the current addressing scheme, these taps can only be stored at the same address on different banks (Fig. 2.13). At the same time, the *Cube* will need to read these taps in different cycles. To solve this limitation, we propose to enhance the addressing capabilities of *L0A* so that different rows of different banks can be accessed with a single memory operation, with a *diagonal* write mode. This modification allows de-facto to perform data scattering

on-the-fly. As reported in Section 2.5.2, this change has a negligible area and power overhead.

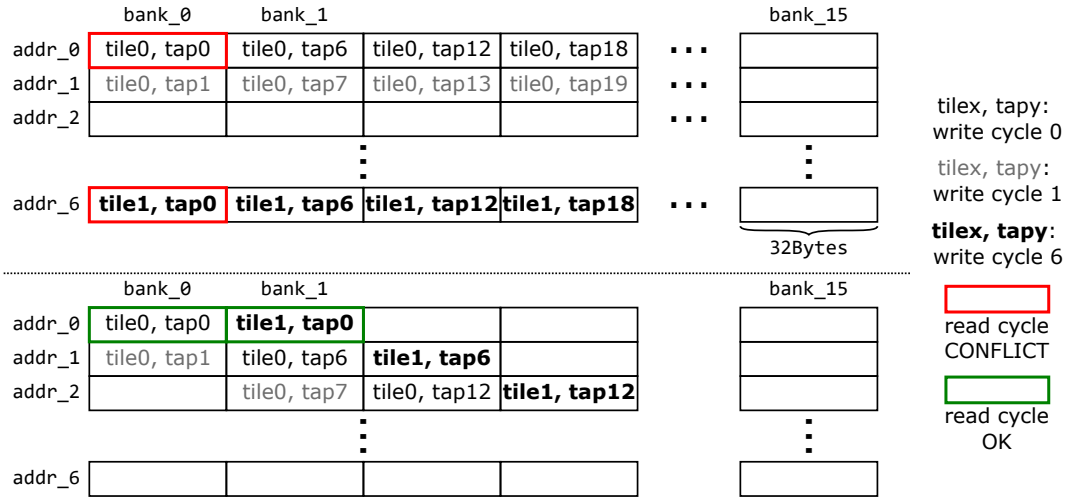


Fig. 2.13 (**top**) Bank conflict arising when using a traditional addressing scheme. (**bottom**) Proposed diagonal access scheme.

To keep the overall pipeline busy, the transformations and MatMuls on the input tiles need to be done in parallel with the output transformation and re-quantization of the previous tile (lines 16-25). For the output transformation engine, the selection of the engine type is mostly driven by the necessity of reducing the number of memory accesses to *L0C*, which are more power-needing because of the larger size of the memory and the higher bitwidth of the values. Since the tap-by-tap engine necessitates multiple accesses per data, it is not a feasible choice, leading us to rely on the row-by-row engine. The available *L0C* bandwidth allows the engines to simultaneously perform up to 16 transformations along the output channels dimension. Therefore, a volume of $C_{out} \cdot \frac{H}{4} \cdot \frac{W}{4}$ tiles in the Winograd domain will be transformed back to the spatial domain in $\frac{C_{out}}{16} \cdot \frac{H}{4} \cdot \frac{W}{4} \cdot 10$ or $\frac{C_{out}}{16} \cdot \frac{H}{4} \cdot \frac{W}{4} \cdot 6$ cycles, by the slow and fast version of the row-by-row engine respectively. At the same time, the *Cube* produces the same amount of data in $\frac{C_{out}}{16} \cdot \frac{Ci}{32} \cdot \frac{H}{4} \cdot \frac{W}{4} \cdot \frac{1}{16} \cdot 36$ cycles. By imposing the *Cube* production rate to be lower or equal to the output transformation engines consumption rate, we obtain that at least 3 ($C_{in} = 96$) and 6 ($C_{in} = 192$) fractal input channel tiles, for the fast and slow engines respectively. Since most of the SOTA networks' layers have less than 192 but more than 96 input channels, we opt for the fast engine.

As for the *L0A* data scattering problem, the *Cube Unit* writes data in *L0C* with the same taps of different tiles in contiguous locations. To perform the output transformations, the output engine needs to access different taps of the same tiles, which will be scattered across different locations in the memory. Thus, the addressing logic is modified as in *L0A* to perform data gathering on the fly.

Lastly, the processing times of all core-level processes must match the load and store operations from and to the *GM*. The weights transformation engine's throughput must match the external bandwidth, as it reads the weights from the *GM* and transforms them on the fly (lines 2-6). Since the external bandwidth is much lower than the in-core bandwidth, we choose for the weight transformations the tap-by-tap engine. Beyond matching the bandwidth, the tap-by-tap engine produces the output data with the data layout needed by the *Cube*, i.e., the taps are already scattered. When reading the weights from the *GM*, the engine would need to perform a gathering operation, which can however be avoided as the layout of the weights in the *GM* can be reorganized offline without any overhead.

A single AI core needs to read at least $18 \cdot 18 \cdot C_{in}$ B of iFMs and to write $16 \cdot 16 \cdot 64$ B of oFMs to match the core throughput. With two available AI cores, as in our system, this constraint doubles, resulting in a required *GM* bandwidth of $\approx 2 \cdot 72 + \frac{7281}{C_{in}}$B/cycle assuming a peak compute efficiency for the core-level operations (lines 13-26). This condition is hard to meet with an AI core clock frequency in the order of hundreds of MHz. Thus we apply three dataflow and system-level optimizations:

1. The necessary bandwidth can be almost halved by sharing the iFMs between the two cores since they operate on different sets of output channels. To achieve this, a *Broadcast Unit* (*BU* in Fig. 2.7) links the cores to the memory controllers. The *BU* can either process broadcast requests in the form of a streaming access pattern [169] or, in a traditional fashion, accept individual memory requests from the MTEs of the two cores and forward them to the memory controllers. In the case of two broadcast memory requests from the cores, the *BU* operates as a DMA and broadcasts data from the *GM* to the *MTEs* of the two cores. The BU has two separate queues for non-broadcast and broadcast requests, with the latter being prioritized to prevent deadlocks.

2. The iFMs volume to be moved can be reduced when the shape is larger than 18×18 by leveraging the halo regions of the 3×3 Conv2d operator with unitary stride.

3. It is possible to decouple read and write operations and prioritize the more critical read transfers by prefetching input tiles and allocating multiple output buffers rather than just two for double buffering.

2.5 Results

2.5.1 Tap-wise Quantization Algorithm

Datasets and Baseline Networks

The datasets used to evaluate the proposed quantization flow are CIFAR-10 [3] and ImageNet ILSVRC [23], two common image classification datasets. CIFAR-10 consists of 60k 32×32 RGB images divided into 10 classes, and ImageNet of 1.4M 224×224 RGB images divided into 1k classes. The datasets are divided into training and validation sets with a split ratio of 90%-10%, respectively. The training set is used for training, and the validation set is for learning rate scheduling. The test sets, used for inference only, consist of 10k and 100k images for CIFAR-10 and ImageNet, respectively. The preprocessing methods applied are random horizontal flip and color normalization for CIFAR-10, and resize, random crop, and color normalization for ImageNet.

For CIFAR-10, we re-implement ResNet-20 [6] and train it from scratch, achieving a baseline accuracy of (94.4%). We also test a light-version of VGG [170], also used by Liu et al. [141] and Lance et al. [144], replacing all but the last dropout layers with batch normalization layers and thus obtaining an accuracy of 92.2%. For the ImageNet dataset, we benchmark ResNet-34 and ResNet-50 from the Torchvision model zoo, with a 72.6%/ 75.5% Top-1 and 90.7%/92.6% Top-5 accuracy on the test set.

The networks are trained in the PyTorch framework [40] applying the Winograd-aware training [134], extended with the tap-wise quantization support.

Tap-Wise Quantization Evaluation

Table 2.2 Ablation study for ResNet-34 on ImageNet

Alg.	WA	$\odot$	2^x	$\nabla_{\log_2 t}$	KD	int$n$	Top-5	Top-1	Δ
im2col						FP32	90.7	72.6	0.0
im2col						8	90.7	72.6	0.0
F_2	✓					8	90.1	71.4	-1.2
F_2	✓					8/10	90.7	72.6	0.0
F_4	✓				✓	8	81.6	59.0	-13.6
F_4	✓				✓	8/10	89.2	69.1	-3.5
F_4	✓	✓				8	90.1	71.4	-1.2
F_4	✓	✓				8/10	90.6	72.0	-0.6
F_4	✓	✓			✓	8	90.7	72.5	-0.1
F_4	✓	✓	✓			8	89.9	70.9	-1.7
F_4	✓	✓	✓			8/10	90.6	72.1	-0.5
F_4	✓	✓	✓	✓		8	89.6	70.8	-1.8
F_4	✓	✓	✓	✓		8/10	90.4	71.8	-0.8
F_4	✓	✓	✓		✓	8	90.0	70.9	-1.7
F_4	✓	✓	✓		✓	8/10	90.7	72.2	-0.4
F_4	✓	✓	✓	✓	✓	8	90.0	71.1	-1.5
F_4	✓	✓	✓	✓	✓	8/10	90.7	72.3	-0.3

$\odot$: tap-wise quantization, 2^x: power-of-two quant., $\nabla_{\log_2 t}$ training, KD: knowledge distillation, WA: Winograd-aware training.

Tab. 2.2 gives an overview of the accuracy of ResNet-34 on the ImageNet dataset when different training and quantization methods are applied. Starting from the FP32 baseline, we retrain and quantize the network in `int8` format, with the weights and feature maps quantized as shown in Eq. (2.14). As with most neural networks, quantizing to `int8` has no or negligible effect on the overall accuracy. Applying the static Winograd-aware algorithm (Section 2.3.1), we train the Winograd F_2 ResNet34 on ImageNet and achieve 71.4% (-1.2% drop) with 8-bit quantization. Increasing the precision of both weights and feature maps to 10-bit in the Winograd domain only (8/10 notation in the table) restores the full accuracy of the network, as expected, since just 3 bits are required for a bit-true calculation (Section 2.3). On the contrary, training the Winograd F_4 version with the Winograd-aware method and KD leads to an accuracy drop of 13.6% w.r.t. the FP32 baseline. Adding two extra bits in the Winograd domain anyway drops the accuracy at least by 3.5%.

The second section of Tab. 2.2 reports the accuracy of the network when tap-wise quantization is applied with unrestricted quantization scaling factors, i.e., they are represented with FP32 precision and can assume any value. With Winograd-aware

static training and straightforward threshold calibration, the accuracy loss is much lower (-1.2%) w.r.t. the accuracy of the network without tap-wise quantization (-13.6%). The accuracy degradation further improves when relaxing the numerical pressure and allowing 10-bits precision in the Winograd domain, and the best result is obtained when also KD is applied.

As explained in Section 2.3.2, constraining the scaling factors to be power-of-two is preferred, as the re-quantization and de-quantization in the Winograd domain become simple shift operations. The third section of Tab. 2.2 summarizes the accuracy results when the scaling factors are limited to power-of-two values. Simply applying the Winograd-aware algorithm with power-of-two tap-wise quantization leads to a 1.7/0.5% drop with `int8` and `int8/10`. Applying knowledge distillation to this scheme brings little or no improvement. Applying $\log_2$ gradients has no positive effect on the accuracy; in fact, a worse performance than the straightforward calibration method is observed due to convergence issues. However, adding KD to $\log_2$ gradients leads to the best results, with an accuracy of 71.1% (-1.5%) for `int8` and 72.3% (-0.3%) for `int8/10`. This result is due to the stabilizing effect of knowledge distillation, which acts as an implicit regularizer [171].

By furtherly investigating the values learned by the network for the tap-wise scaling factors, we see that the feature maps are right-shifted in a range from 1 to 5 bits, (i.e., $s \in \{2, 2^2, \dots, 2^5\}$) and the weights from 2 to 10 bits. The wide range assumed by the scaling factors clearly demonstrates why quantizing with a single scalar for all the taps leads to a huge accuracy drop.

Comparison with Other SOTA Winograd-Aware Quantization Methods

There have been a variety of Winograd-aware quantization methods recently presented. Tab. 2.3 gives a full overview of the main methods and a comparison with our solution. In the table, together with the accuracy of the Winograd quantized networks, we also report the baseline accuracy. We will then compare the relative accuracy drop to consider the differences in training and implementation of the different works. The results reported for our solution are obtained by applying the Winograd-aware training method, the powers-of-two tapwise quantization, the $\log_2$ gradients, and knowledge distillation.

Table 2.3 SoA Winograd-aware quantization methods.

CIFAR-10/ResNet-20		int*n*	Top-1	Ref.	Δ
[172] Legendre (static)	F_4	8	85.0	92.3	-7.3
[172] Legendre (static)	F_4	8/9	89.4	92.3	-2.9
[172] Legendre (flex)	F_4	8	91.8	92.3	-0.5
[172] Legendre (flex)	F_4	8/9	92.3	92.3	**0.0**
[134] Winograd-Aware (s)	F_4	8	84.3[1]	93.2	-8.9
[134] Winograd-Aware (f)	F_4	8	92.5	93.2	-0.7
[140] Winograd AdderNet	F_2	8	91.6	92.3	-0.7
(ours) Tapwise Quant.	F_4	8	93.8	94.4	-0.6
(ours) Tapwise Quant.	F_4	8/9	94.4	94.4	**0.0**
CIFAR-10/VGG-nagadomi		int*n*	Top-1	Ref.	Δ
[141] Sparse	F_2	FP32	93.4	93.3	0.1
[144] Quant. Winograd	F_2	8	90.3	90.4	-0.1
(ours) Tapwise Quant. (static)	F_4	8	90.8	92.0	-1.2
(ours) Tapwise Quant. (static)	F_4	8/9	91.9	92.0	-0.1
(ours) Tapwise Quant. (static)	F_4	8/10	92.0	92.0	**0.0**
ImageNet/ResNet-50		int*n*	Top-1	Ref.	Δ
[145] Complex Numbers	F_4	8	73.2	73.3	-0.1
[146] Residue Numbers	F_{14}	8	75.1	76.1	-1.0
[147] LoWino	F_4	FP32/8	75.5	76.1	-0.6
(ours) Tapwise Quant. (static)	F_4	8	75.2	75.5	-0.3
(ours) Tapwise Quant. (static)	F_4	8/10	75.5	75.5	**0.0**

[1]Not reported. Reproduced with the open-source code [134].

The first part of Tab. 2.3 reports the results of ResNet-20 on CIFAR-10. Without adding any computational overhead, we improve the accuracy of the static Winograd-aware (WA) method [134] from 84.3% to 94.4%, i.e., the same accuracy of the baseline FP32 model. In [134], a *flex* methodology is also proposed, with which the transformation matrices are included in the set of trainable parameters. We outperform the flex WA method [134] by 1.9%. Moreover, it has to be noted that, with this technique, the transformation matrices to be used for each layer at inference time are different. If designing a custom HW accelerator, this prevents exploiting the fact that the transformation matrices are known a priori to create efficient transformation engines, as in Section 2.4.2. [172] proposes to perform the Winograd transformations in Legendre polynomial base instead of canonical base. Then they apply the same *static* and *flex* quantization methodology proposed in [134]. We outperform the Legendre-F_4 method 2.1%. It has to be noted that, to move from

canonical to Legendre polynomial base, the number of transformations doubles, making the methodology not HW-friendly.

The second part of the table shows the results for the light version of VGG (VGG-nagadomi [170]) trained on the CIFAR-10 dataset. Our baseline network achieves a 92.0% accuracy, and the tap-wise powers-of-two quantized F_4 shows a drop of 1.2%, 0.1% and no drop for `int8`, `int8/9`, and `int8/10` respectively. No previous work reports the accuracy for a F_4 implementation for this particular network, so we will compare to the F_2 results. To reduce the computational intensity further, Liu et al. [141] adopt a technique orthogonal to quantization pruning the weights in the Winograd domain and obtaining a zero drop on their baseline accuracy in FP32 precision. Li et al. [144] propose to execute the Winograd transformations in FP32 precision and then quantize to `int8` once in the Winograd domain, achieving a 0.1% accuracy reduction. While keeping part of the operations in FP32 precision clearly helps to reduce accuracy loss, it deviates from the main goal of making quantized-Winograd HW-friendly.

Finally, the last section of the table reports the results for ResNet-50 trained on ImageNet. We achieve 75.2%/92.3% (-0.3%/-0.3%) with 8 bits and 75.5%/92.5% (0.0%/-0.1%) when extending the precision in the Winograd domain to 10 bits (`int8/10`). On this benchmark, we can compare to Meng et al. [145], Liu et al. [146], and LoWino [147]. Meng et al. [145] uses complex root points to obtain numerically more stable but complex transformation matrices that quantized to `int8` lead to a small accuracy drop (-0.1%, but with a lower baseline accuracy). Liu et al. [146] uses the residue number system (RNS) and very large tile size, i.e., 14×14, to compensate for the transformation overhead introduced. The accuracy decreases by 1% from a 76.1% baseline accuracy, even though the RNS could perform the `int8` operations losslessly (transformations and elementwise multiplications). This may be due to the quantization of the transformation matrices and the high numerical error introduced by the larger tile size [139, 52]. In LoWino [147], the weights and feature maps are kept in FP32 representation but quantized linearly to 8 bits before and after the elementwise multiplication in the Winograd domain. They achieve an identical accuracy of 75.5% as our method with int8/10, although with an accuracy drop of 0.6% instead of 0% as they start from a higher baseline. Moreover, while obtaining the same reduction in operation count, LoWino requires a $4\times$ higher bandwidth than our solution, eliminating any benefit of the Winograd F_4 approach as

shown in Section 2.5.2. Notably, only our solution with `int8/10` can obtain zero accuracy drop with ResNet-50 on the ImageNet dataset.

Tap-wise vs. Channel-wise Quantization

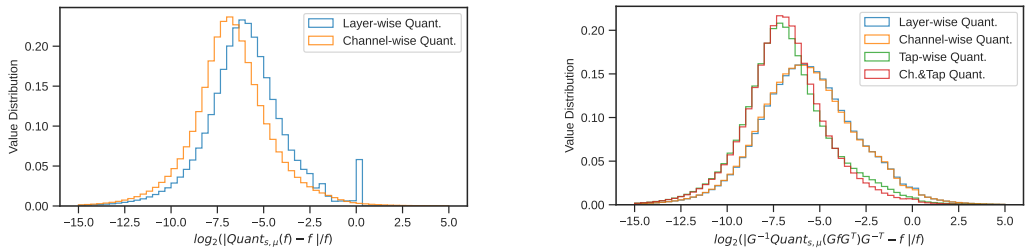
Previous studies have demonstrated that the accuracy of quantized networks can be considerably increased using fine-grain quantization methods, notably (output) channel-wise quantization [173, 174]. Therefore, we discuss the differences between the tap-wise and channel-wise quantization strategies in this section. We evaluate the quantization error on the weights of a pre-trained ResNet-34 from the Torchvision model zoo, although a similar trend can also be observed for the feature maps.

The scaling factors s are determined as follows:

$$\hat{\gamma} = \arg\min_{\gamma} \sum_{f} |\text{Quant}_{\mu,s}(f) - f|/|f|, \quad s = \gamma\sigma/2^{n-1}, \tag{2.26}$$

where $Quant_{s,\mu}(x) = \mu + s\lfloor (x-\mu)/s \rceil_{\texttt{int}n}$, the mean μ, the standard deviation σ, and the optimized scaling factor $\hat{\gamma}$ are obtained per layer (uniform quantization strategy), per channel, or per tap.

Fig. 2.14a shows the distribution of relative quantization error (in log2 scale) of all layers with kernel size 3×3 for a uniform and for a channel-wise quantization strategy in the spatial domain for $n = 8$ bits. Channel-wise quantization reduces the mean relative error from $2^{-6.01}$ to $2^{-6.72}$ ($1.7\times$ reduction).



(a) Spatial Quantization

(b) Quantization in Winograd domain

Fig. 2.14 Quantization error distribution for the weights in (a) spatial and (b) Winograd domain on ResNet-34 using different strategies: layer-wise quantization, channel-wise quantization, tap-wise quantization, and channel- & tap-wise quantization.

Fig. 2.14b shows the error distribution for layer-wise, channel-wise, and tap-wise quantization in the Winograd domain. To compare the error caused by the quanti-

zation, we quantize in the Winograd domain ($Quant(GfG^T)$). Then we transform the data back to the spatial domain by calculating the Moore-Penrose inverse of the transformation matrices based on singular value decomposition (SVD). In this situation, the channel-wise quantization reduces the mean relative error significantly less, from $2^{-5.58}$ to $2^{-5.62}$. On the other hand, tap-wise quantization shows much better performance with a 2.3× reduction of the mean error. The channel-wise and tap-wise quantization combination further improves the average error by 1.06× but at the cost of a much more complicated computational phase. Therefore, this combined quantization strategy may be worthy only for networks with significantly spread channel distribution.

2.5.2 System Evaluation

Experimental Setup

Area and Power. We developed the RTL of the components of the AI core that are most impacted by the Winograd extensions, namely the *Cube Unit*, the *MTE1*, and the *FixPipe* module, to evaluate the area and power consumption of the accelerator.

The design is implemented with a high-*k* metal gate (HKMG) 28 nm CMOS technology, a corresponding multi-VT standard cell library, and a supply voltage of 0.8 V in typical corner. We have synthesized, placed, and routed the design, closing it at a clock frequency of 500 MHz in typical operating conditions. We used an industrial-grade memory compiler for the SRAM and register file macros. For the timing-annotated (standard delay format, SDF) post-place & route gate-level simulations, we have picked input data segments from the first 3×3 layer of a ResNet-34, quantized with our flow. Lastly, the power consumption is estimated based on the extracted switching activities (value change dump, VCD).

Performance Profiling. To profile the overall system on a wider set of benchmarks, we designed an event-based simulator [175]. The simulator models the timing behavior to estimate the throughput of the system, and also the data movements and the computations to check the correctness of the results. The simulator was validated by micro-benchmarking it against the parts of the system developed in RTL. Precisely, we compared the number of cycles estimated by the simulator with that

obtained from the RTL simulator, obtaining a 5% worst-case difference on several small and medium-sized Conv2D operations.

The simulator adopts a simple model for the DRAM subsystem that serves memory requests in order. The completion time of a memory request depends on the maximum bandwidth (81.2 B/cycle), which corresponds to $\approx 0.8 \cdot 51.2$ GB/s given the clock frequency of the core, and on a fixed average latency (150 AI core cycles) with a jitter extracted from a zero-mean Gaussian distribution with a variance of 5 cycles. The memory bandwidth and latency characteristics are chosen to meet the expected performance of an LPDDR4x-3200 memory with two channels [176, 177]. Not using a detailed model of all the DRAM resources (e.g., bank and row-buffer conflicts, channel bandwidth, command scheduling) has minimal effect on the performance of the cases under analysis [175], as the memory accesses are regular and follow a streaming pattern.

To model the energy consumption in the simulator, we project the power consumption of the memories and the computational units obtained from the back-annotated gate-level simulations. To take into account the additional logic in $L1$ needed to handle bank conflicts and arbitration of read and write ports, we multiply the values obtained from the memory compiler for $L1$ area and energy cost by a $1.5\times$ factor.

Workloads. To evaluate the system performance, we adopt two sets of benchmarks:

1. A synthetic benchmark suite of 63 3×3 Conv2D layers with common values for batch size (B), number of input and output channels (C_{in}, C_{out}), height (H), and width (W) of the oFMs.

2. A benchmark suite with the Conv2D layers of 7 state-of-the-art CNN networks to evaluate the energy savings and the speed-up on models having different architectures.

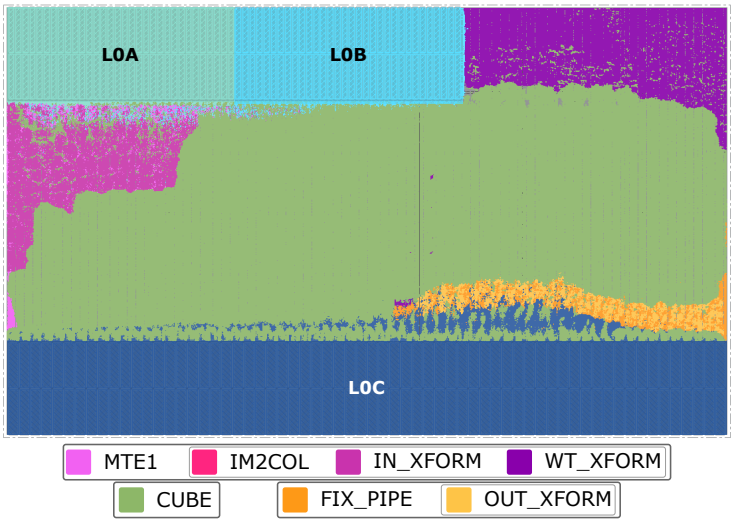
ResNet-34 and ResNet-50 [6] are taken as representative of computationally intensive networks for the classification tasks; UNet [132] for high-resolution semantic segmentation tasks; RetinaNet-ResNet50-fpn [178], YOLOv3 [131], and SSD-VGG16 [130] for object detection tasks. The networks were taken from the Torchvision Python model zoo.

Area and Power Analysis

Table 2.4 AI Core breakdown at 0.8 V and 500 MHz. Power consumptions marked with * refer to the Im2col kernel, or with † to the F_4 Winograd kernel. The cube TOp/s/W reported for the F_4 Winograd kernel are computed using the equivalent TOp in the spatial domain, i.e., 4× the TOp of the CubeUnit.

Unit		Area	Peak Power	TOp/s/W
Cube		2.04 mm² (19.2%)	*1521 mW †1923 mW	*5.39 +17.04
MTE1	Im2col	0.03 mm² (0.3%)	30 mW	
	IN_XFORM	0.23 mm² (2.2%)	145 mW	5.3
	WT_XFORM	0.32 mm² (3.0%)	228 mW	1.6
FIX_PIPE	OUT_XFORM	0.10 mm² (0.9%)	114 mW	2.3

Memory	Size	Area	Rd Cost	Wr Cost
L0A	64 kB	0.32 mm² (3.1%)	0.22 pJ/B	0.24 pJ/B
L0B	64 kB	0.32 mm² (3.1%)	0.22 pJ/B	0.24 pJ/B
L0C - PortA	288 kB	1.24 mm² (11.7%)	0.23 pJ/B	0.29 pJ/B
L0C - PortB			*0.31 pJ/B +0.69 pJ/B	- -
L1	1248 kB	5.97 mm² (56.1%)	0.92 pJ/B	0.68 pJ/B



Tab. 2.4 provides information on the physical layout of the implemented hardware extensions, as well as the area and power breakdown of the AI core. As expected, the *Cube Unit* dominates both the area and the power consumption of the computational

modules, being at least 6.4× bigger and 6.7× more power-demanding than a single Winograd engine.

Overall, just 6.1% of the core area is taken up by all of the Winograd transformation engines. When analyzing the reported power costs, it has to be noted that most of the time, only the input and output transformation engines are simultaneously active, while the power cost of the weight transformation engine is amortized over the computations of all activations. Therefore, the Winograd extension adds ≈ 17× of power overhead to the *Cube Unit*, but it also reduces its number of active cycles to one-fourth compared to the im2col.

Tab. 2.4 reports two different power consumptions of the *Cube Unit* and of the *L0C-Port B*, as they respectively increase for the Winograd kernel by 1.26× and 2.22×. This is due to the higher switching power consumption caused by the lower sparsity of weights and activations in the Winograd domain [141]. Nevertheless, utilizing the Winograd kernel rather than the im2col results in a compute datapath that is ≈ 3× more energy-efficient.

On the memory side, both the energy per access and the area are highly related to the memory size. Despite its more complex addressing logic and access pattern (Section 2.4.3), *L0A* has a negligible area and energy access cost overhead w.r.t. *L0B*. On the other hand, the rotation logic required on the output port (*PortB*) of *L0C* has a significant effect on power consumption. However, in practice, the average access cost to *L0C* is much less expensive, as the (*PortB*) of *L0C* is on average less used than (*PortA*).

Throughput Analysis

Tab. 2.5 shows the speed-up of the Winograd operator compared to the im2col for 3×3 Conv2D layer configurations, from which we can identify two macro trends.

Larger resolution or batch size → higher speed-up. As explained in Section 2.4.3, the weights are reused for all the iFMs in a weight stationary fashion. As a result, when weight reuse is low, the performance is constrained by the weights transfer. For example, with 256 input and output channels and batch size equal to 1, the speed-up increases from 1.98× to 3.30× when the resolution increases from 32×32 to 128×128. When changing the batch size from 1 to 8 at iso-resolution (32×32) it increases from 1.98× to 3.18× .

Table 2.5 Throughput of the Winograd operator normalized to the im2col operator for different 3×3 Conv2D layers with stride equals to 1 and padding *same*. H, W refers to the output resolution.

	B	1									8								
	Cin	64		128		192	256		512		64		128		192	256		512	
	Cout	64	128	128	192	256	384	256	512	512	64	128	128	192	256	384	256	512	512
H,W	16	0.99	1.00	1.03	1.13	1.13	1.26	1.03	1.12	0.99	1.30	1.69	2.11	1.77	2.02	2.27	2.38	2.34	2.00
	32	1.15	1.27	1.70	2.08	2.21	2.37	1.98	2.02	1.59	1.31	1.84	2.62	2.24	2.59	2.93	3.18	3.16	2.48
	64	1.34	1.73	2.50	2.12	2.44	2.75	2.96	2.93	2.29	1.28	1.87	2.66	2.27	2.65	3.07	3.37	3.36	2.65
	128	1.27	1.84	2.64	2.25	2.62	3.02	3.30	3.29	2.59	1.25	1.84	2.68	2.29	2.68	3.11	3.42	3.42	2.69

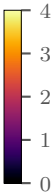


Fig. 2.15 displays the cycle usage breakdown of the critical path of the Winograd operator normalized to the im2col (hatched bar) to more clearly illustrate the bottlenecks for various workloads. In particular, looking at the first and the third workload in Fig. 2.15, The normalized percentage of cycles spent in the weight transfer and weight transformations falls from 13% to 2% when the batch size goes from 1 to 8. It should be noted that the weight transformation engine's throughput has been adjusted to match external weight transfers while taking up the least amount of area. By eliminating the weight transformation engine's contribution, another important path will emerge where the weight data transfer replaces the transformations. This analysis also demonstrates the necessity for on-the-fly weight transformation rather than reading the transformed weights from the external memory. The load overhead would be substantially higher and more challenging to amortize due to the Winograd domain's 4× weights expansion factor.

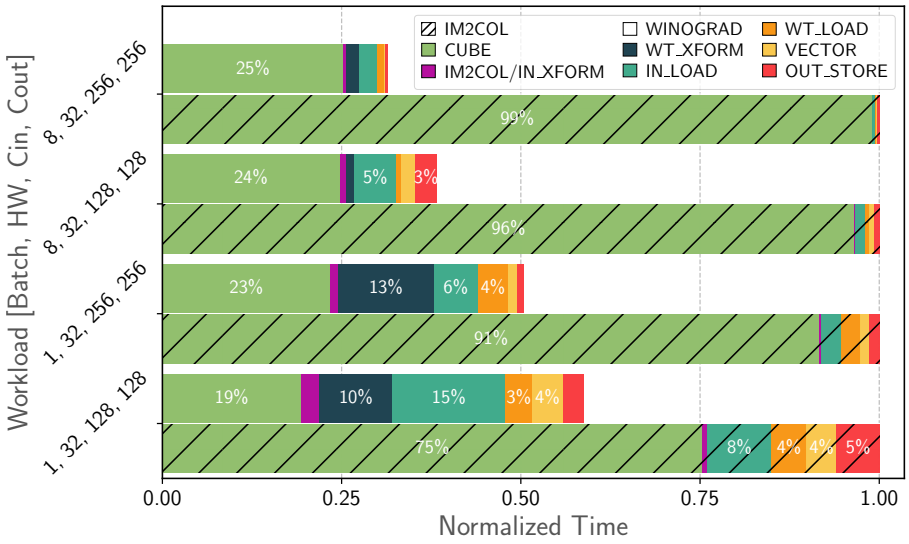


Fig. 2.15 Cycle Breakdown for im2col vs. Winograd F_4.

Larger number of input channels → higher speed-up. The output reuse opportunity within the core is higher with a larger number of input channels, having the effect of reducing the bandwidth occupied by the store operations of the oFMs. As a result of the data reuse, more bandwidth is available for the transfer of the iFMs, which has a notable impact on the performance since the iFMs are broadcasted to the two cores. For example, with batch size equal to 8, spatial resolution equal to 32×32, and output channels equal to 256, increasing the number of input channels from 128 to 256 leads to a speed-up increase from 2.62× to 3.18×. In Fig. 2.15, the higher bandwidth to dedicate for the iFMs transfer translates to a reduction of the cycles occupied by the MTE2 from 5% to 2% for the first two workloads and from 15% to 6% for the last ones. In fact, the main reason why the F_4 Winograd operator fails to deliver the expected 4× speed-up on our system is bandwidth limitation. The input and output transformation engines, as seen in Fig. 2.15, never constitute the operator's bottleneck because their throughput is sized to precisely match the input and output data rate of the Cube Unit.

Comparison with NVDLA

Table 2.6 Comparison of NVDLA and our accelerator system.

	8× F2 NVDLA		**8× F2 NVDLA**		**F4 OURS**	
Bandwidth[1]	128 Gword/s		42.7 Gword/s		41 Gword/s	
Peak Throughput	8 TOp/s		8 TOp/s		8 TOp/s	
B, H, W, C_{in}, C_{out}	t [μs]	SU [×]	t [μs]	SU [×]	t [μs]	SU [×]
8, 32, 32, 128, 128	79.1	2.03	106.2	1.74	59.8	**2.62**
8, 32, 32, 128, 256	144.7	2.13	175.8	1.89	118.7	**2.59**
8, 32, 32, 256, 512	574.6	2.09	1736.5	0.72	383.7	**3.16**

[1]Word-bandwidth to external memory: 1 word is 2 Bytes for FP16 (NVDLA) and 1 Byte for INT8 (ours). We compare NVDLA with quasi-infinite bandwidth (i.e., 256 GB/s) and iso-word-bandwidth.

In Tab. 2.6, we compare our accelerator to the open-source NVDLA accelerator v1, supporting direct convolution (in FP16 and INT8) and Winograd F_2 convolution (FP16 only) with an on-chip memory of 512 kB per engine [154]. We modified the Linux driver used in the virtual platform in order to write out the sequence of reads

and writes from/to the control and status registers of the accelerator, which we then use to simulate the RTL for performance benchmarking. This is because NVDLA does not provide any tools to convert a model that can be used with its compiler into a format accepted by its verification infrastructure. The results are compared with the expected values to verify the functional correctness.

The results are summarized in Tab. 2.6. A single NVDLA core has a peak throughput of 1 TOp/s at 1 GHz, therefore, to match the peak throughput of our system (8 TOp/s), we use 8 NVDLA engines. We consider two different configurations for the NVDLA-based system: the column on the left in Tab. 2.6 describes a system with quasi-infinite bandwidth, while the system referred to by the middle column has 42.7 Gword/s, i.e., 85.4 GB/s in FP16, to match the more realistic bandwidth constraints of our system, i.e., 41 Gword/s (41 GB/s with INT8 for our system, 82 GB/s with fp16 for NVDLA). As the public NVLDA version only supports FP16, we utilize words rather than bytes for the iso-bandwidth evaluation because it is reasonable to assume that performance scales with word width. Our accelerator surpasses NVDLA by 21 to 50%, even if NVDLA with quasi-infinite bandwidth comes close to the theoretical 2.25× speed-up. Winograd convolutional algorithm on NVDLA becomes substantially memory-bound in the more realistic case of the system with constrained external bandwidth, significantly lowering its advantages over the direct convolutional approach. One of the causes of this degradation is that NVDLA requires offline weight transformation, which raises the volume of transferred weight by $4^2/3^2 = 1.78\times$. Furthermore, the Winograd kernel performs even worse than the direct convolution in situations when the input feature maps of a single layer must be repeatedly transferred from external memory because they cannot be entirely kept on-chip. Overall, leveraging Winograd F_4 vs. F_2, on-the-fly weight translation, and higher utilization, our accelerator system performs between 1.5 and 3.3× faster than NVDLA at the same peak throughput and external bandwidth.

Full Network Evaluation

Tab. 2.7 reports the evaluation of various state-of-the-art CNNs run on the proposed system. The throughput of the 3×3 compute-heavy Conv2D layers increases by 1.9× on average and up to 2.60× when the F_4 Winograd operator is used. The improvement in network throughput is dependent on the particular architecture. In fact, compared to networks dominated by 3×3 convolutions, like U-Net or YOLOv3,

Table 2.7 Throughput and energy efficiency evaluation. Values in parentheses refer only to the Winograd layers. The speed-up columns marked with the symbol * refer to a system with a higher external memory bandwidth (1.5×).

			Throughput [Imgs/s]										Energy Eff.[Inf/J]
Network	**Batch**	**Res.**	**im2col**	F_2	F_4	F_2 **vs. im2col**	F_4 **vs. im2col**	F_4 **vs.** F_2	$^*F_2$ **vs. im2col**	$^*F_4$ **vs. im2col**	$^*F_4$ **vs.** F_2	F_4 **vs. im2col**
ResNet-34	1	224	921	950	985	1.03x (1.29x)	1.07x (1.39x)	1.04x (1.08x)	1.07x (1.15x)	1.10x (1.52x)	1.03x (1.32x)	1.15x (1.79x)
ResNet-50	1	224	669	676	684	1.01x (1.29x)	1.02x (1.37x)	1.01x (1.06x)	1.02x (1.14x)	1.03x (1.51x)	1.01x (1.32x)	1.05x (1.79x)
RetinaNet-R-50	1	800	38	51	57	1.34x (1.73x)	1.49x (2.18x)	1.11x (1.26x)	1.40x (1.79x)	1.60x (2.43x)	1.14x (1.36x)	1.51x (2.00x)
SSD-VGG-16	1	300	162	243	252	1.50x (1.59x)	1.55x (1.89x)	1.03x (1.19x)	1.58x (1.75x)	1.71x (2.05x)	1.08x (1.17x)	1.70x (2.03x)
UNet	1	572	46	75	81	1.62x (1.71x)	1.74x (2.18x)	1.07x (1.27x)	1.75x (1.85x)	2.00x (2.49x)	1.14x (1.35x)	1.85x (2.28x)
YOLOv3	1	256	317	349	358	1.10x (1.34x)	1.13x (1.46x)	1.03x (1.09x)	1.15x (1.47x)	1.16x (1.41x)	1.01x (0.96x)	1.43x (2.23x)
YOLOv3	1	416	154	188	195	1.22x (1.66x)	1.27x (1.85x)	1.04x (1.11x)	1.27x (1.77x)	1.35x (1.83x)	1.06x (1.03x)	1.35x (1.92x)
SSD-VGG-16	8	300	176	304	328	1.68x (1.71x)	1.83x (1.97x)	1.09x (1.15x)	1.74x (1.77x)	2.06x (2.26x)	1.18x (1.28x)	1.78x (1.90x)
YOLOv3	8	256	496	664	680	1.33x (1.72x)	1.37x (2.40x)	1.03x (1.40x)	1.42x (1.87x)	1.51x (2.32x)	1.06x (1.24x)	1.50x (2.57x)
ResNet-34	16	224	1472	1776	2000	1.22x (1.73x)	1.36x (1.93x)	1.11x (1.12x)	1.24x (1.52x)	1.46x (2.29x)	1.18x (1.51x)	1.40x (2.03x)
ResNet-50	16	224	816	848	880	1.05x (1.73x)	1.07x (1.90x)	1.02x (1.10x)	1.06x (1.51x)	1.10x (2.25x)	1.04x (1.49x)	1.13x (2.02x)
YOLOv3	16	256	480	672	672	1.38x (1.92x)	1.38x (2.60x)	1.00x (1.35x)	1.44x (2.01x)	1.51x (2.46x)	1.05x (1.22x)	1.51x (2.59x)

the Winograd algorithm's advantages are less significant for networks with a lot of 1×1 convolutions, like ResNet-50. The Winograd algorithm, however, excels when the batch size or input resolution are increased. In the case of ResNet-34, for instance, utilizing a batch size of 16 instead of 1 results in a speedup that is increased from 1.07× to 1.36×. The speedup on SSD-VGG-16 goes from 1.55× to 1.83× when the batch size is increased, which is even more impressive.

In Tab. 2.7, we also provide throughput information for the F_2 Winograd operator, which was constructed using the same methods as F4 in Section 2.4.2 and the same dataflow as F_4 in Listing 2.1. F_2 and F_4 yield comparable performance when the 2.25× computational reduction introduced by the F_2 operator makes the workloads of the layers memory-bound, though the F_4 configuration always beats the F_2. However, F_4 enhances the throughput compared to F_2 by up to 1.4× when the batch size or input resolution are increased, especially for particularly compute-intensive networks like SSD-VGG-16, YOLOv3, and UNet. Tab. 2.7 additionally displays the speed-up relative to the im2col algorithm for a system with a greater bandwidth in order to illustrate the advantages of the Winograd F_4 algorithm (1.5×, i.e., the ratio between a DDR5 and a DDR4 memory). In this scenario, whereas Winograd F_2 reaches a speed-up plateau of about $\approx 1.8\times$, Winograd F_4 takes use of the extra external memory bandwidth to double end-to-end throughput in comparison to the baseline.

Winograd F_4 generally outperforms im2col and F_2, even if the gains over F_2 are not always significant and heavily reliant on the shapes of the individual network layers. In particular, the introduction of the Winograd transformation engines limits the possibility of applying loop transformations, e.g., blocking and reordering, to the outer loops of the convolution operation, beyond constraining the dataflow within a single AI core. Additionally, the spatial resolution of the output activation tiles must be a multiple of 4, which restricts the available tiling factors and, in some circumstances, introduces the necessity of zero-padding and adds inefficient computations. These extra limitations have an impact on how often data is reused in the core and how frequently external memory is accessed, further highlighting the bandwidth constraints. Further proof is given by the layer-wise analysis in Tab. 2.7, which reveals that not the same layers of the network are mapped either on Winograd F_2 or on Winograd F_4 depending on the available extension. For example, for YOLOv3 with an input resolution of 256 and batch size 1, the Winograd F_2 outperforms Winograd F_4 because it is used to process the deep layers of the network where the small spatial resolution ($\leq 16 \times 16$) makes the Winograd F_4 perform worse than the im2col algorithm. However, for the YOLOv3 with an input resolution of 256 and batch size 8, Winograd F_4 results in a $1.4\times$ higher throughput than Winograd F_2 (i.e., with DDR4)

Even if the throughput increase may be lower than the theoretical $4\times$, Winograd F_4 nevertheless decreases the use of the MatMul engine, which is typically the most power-hungry computing resource. As a result, the overall energy efficiency increases, as examined in the paragraphs that follow. Therefore, accelerator designers can use the methodology described in Section 2.4.2 to develop the transformation engines for Winograd F_2 and integrate them with the Winograd F_4 ones, depending on the application use cases and the area budget. This will enable the compiler to choose the best computational kernel for each layer of the network.

Fig. 2.16 reports, on the left, the average number of read and write accesses and, on the right, the average energy breakdown of the Winograd F_4 operator for the Winograd layers of the networks analyzed in Tab. 2.7. All values are normalized to the im2col Conv2D operator. Since the weights are transformed on the fly in the core, the number of read accesses to the weights in *GM* remains unchanged. On the other hand, the weights expansion factor $\frac{(m+2)^2}{9}=4$ caused by the Winograd transformation has the effect of increasing the write accesses to *L1*. In the new system, the *Cube Unit* directly accesses the weights from *L1* rather than using *L0B*

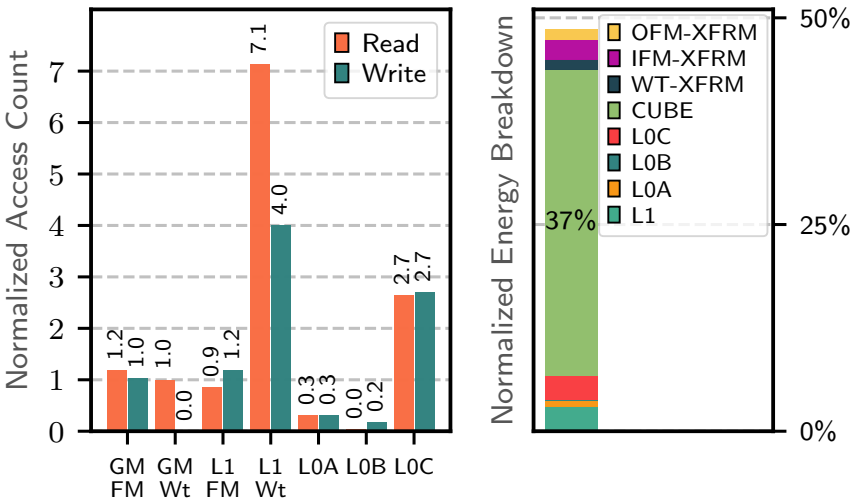


Fig. 2.16 Number of memory accesses (left) and energy breakdown (right) for Winograd F_4 w.r.t. im2col.

as a buffer as the im2col operator. Therefore we witness a significant increase in the read accesses to the weights in *L1*. However, since the F_4 Winograd algorithm reduces by one-fourth the total number of weight reads and the *L1* energy access cost is only $3\times$ higher than that of *L0B*, the overall energy consumption is lower. The accesses to *L0B* are only performed when transforming the weights, so their cost is highly amortized over time. Because of the lower output channel reuse factor (64), the read accesses to the iFMs in *GM* and samely the iFMs write accesses to *L1* slightly increase. The read accesses to the iFMs in *L1* and the write accesses to *L0A* decrease since the Winograd transformation increases the volume of the iFMs only by a factor of $\frac{(m+2)^2}{m^2}$=2.25 for $m=4$, while the im2col has an expansion factor 9 for the 3×3 convolution. The number of read accesses to *L0A* decreases proportionally to the reduction of the *Cube Unit* active cycles. Since the oFMs are in the Winograd domain, the number of read and write accesses to *L0C* is higher.

Overall, the energy the memory subsystem uses is comparable between F_4 Winograd and the im2col operator. Still, the Winograd F_4 algorithm uses less energy overall by more than $2\times$ thanks to its reduction of the active cycles of *Cube Unit*, which, as can be seen in Fig. 2.16, accounts for the majority of the core's energy usage.

This analysis reveals yet another significant benefit of the Winograd F_4 algorithm over the im2col and the Winograd F_2 algorithm: even though the theoretical $4\times$

MACs reduction may not always translate into an equivalent wall-clock time speed-up, it guarantees higher energy efficiency, making it an ideal fit for inference DSAs.

2.6 Conclusion

This chapter focused on the tap-wise quantization algorithm to enable efficient quantized F_4 Winograd convolution. With 8-bits integers for feature maps and weights in the spatial domain and 10-bits integers in the Winograd domain, ResNet-20, VGG-nagadomi, and ResNet-50, with Winograd F_4 Conv layers achieve the same accuracy as the FP32 baselines, on CIFAR-10 and ImageNet respectively. The proposed solution outperforms SOTA integer-only and F_4-aware quantization techniques on all evaluated networks and tasks. Moreover, we introduced custom HW extensions integrated into ad industrial-grade AI accelerator to efficiently process integer Winograd F_4 layers. Thanks to the higher computational reduction of F_4, the higher resources utilization obtained with the optimized dataflow, and the tuned bandwidth requirements for the on-the-fly transformations, our proposed system outperforms NVDLA and its Winograd F_2 extension by 1.5 to 3.3$\times$ for the same compute throughput and bandwidth constraints. The hardware extensions have a small area (6.1% of the core area) and power (17% compared to the MatMul engine) overhead over the baseline architecture and achieve up to 3.42$\times$ speed-up on compute-intensive convolutional layers. An exhaustive analysis of numerous cutting-edge computer-vision benchmarks showed up to 1.83$\times$ faster end-to-end inference and 1.85$\times$ better energy efficiency.

Chapter 3

Hardware-Efficient Capsule Networks

Part of the work described and of the figures appearing in this chapter has been previously published in [179–181].

3.1 Introduction and Motivation

In the pursuit of improving the learning capabilities and accuracy of CNNs, a novel architecture called *capsule network* (CapsNet) was introduced [5]. Hereon, we will use the term CapsNet to identify any capsule network and ShallowCaps for the specific architecture proposed in [5]. These networks aim to learn the hierarchical and spatial information of the input features similarly to the human brain's functionality, according to our current understanding. Single neurons are substituted with capsules, i.e., vectors of neurons, that encode an entity's instantiation probability and its main features. To overcome the loss of information introduced by pooling layers, the pooling operation is substituted by a dynamic routing process between the capsules of adjacent layers.

The drawback of capsule networks is that they create an additional dimension w.r.t. the tensors of the convolutional and fully connected layers of traditional convolutional neural networks. This makes them more challenging in terms of memory requirement, memory bandwidth, and computational workload, putting more pressure on the underlying hardware used for deployment [182]. As a demonstration, Tab. 3.1 reports the computational intensity measured by the ratio between MAC operations and weights memory for three networks, LeNet [4], AlexNet [13], and

ShallowCaps [5]. The compute-intensive nature of ShallowCaps is evident, and it is attributed to the larger dimension of the constituent elements of the CapsNets and the high computational effort required to dynamically route the capsules.

Table 3.1 Computational-intensity comparison between different DNN models.

Architecture	**MAC/Mem Ratio**
LeNet [4]	5.68
AlexNet [13]	19.40
CapsNet [5]	**31.93**

In addition to the analysis of Tab. 3.1, which measures the compute intensity in terms of multiplications and additions, it is also necessary to consider the impact of the more complex activation functions used in capsule networks, i.e., the *squash* and *softmax*. To demonstrate the impact of these functions, we perform a detailed analysis of the energy consumption and area footprint of a MAC unit, which is the basic block of DNNs accelerators, and of the specialized hardware blocks that perform the *squash* and *softmax* operations. We design different versions of a MAC unit, a squash module, and a softmax module, varying their bitwidth, and we synthesize them in a UMC 65nm CMOS technology with the Synopsys Design Compiler tool to measure their area and energy consumption. In order to quantify the energy consumption, RTL simulations with randomized inputs generated from a uniform distribution are performed. These simulations generate a Switching Activity Interchange Format (SAIF) file, which is subsequently utilized within the Synopsys Design Compiler tool for energy estimation. Fig. 3.1 and 3.2 show how the area and energy consumption of MAC, squash, and softmax units diminish with decreasing bitwidth. As expected, the squash and the softmax functions require more energy and area than a simple MAC operation.

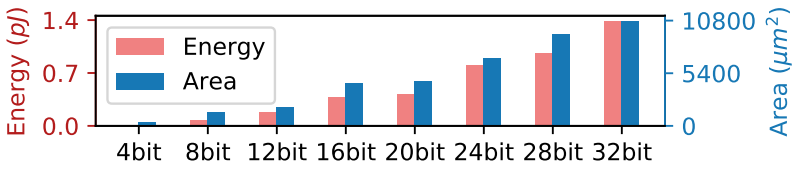


Fig. 3.1 Energy consumption and area footprint for a fixed-point multiply-and-accumulate unit (MAC) with different bitwidths.

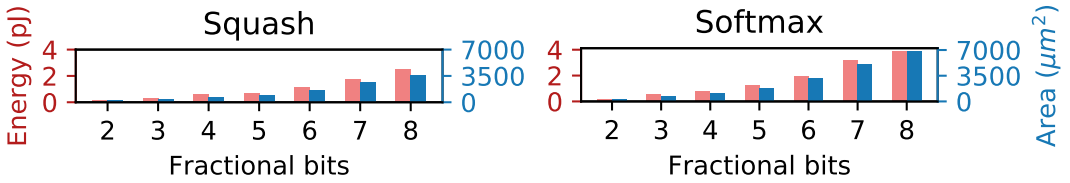


Fig. 3.2 Energy consumption and area footprint for fixed-point modules performing (**left**) the squash and (**right**) the softmax with different bitwidths.

Despite the promising results of capsule networks in terms of accuracy, their architectural and computational complexity has been a deterrent to both the development of new models and their hardware deployment. Therefore, our overarching goal is to push the exploration of new hardware-efficient capsule network models with a set of orthogonal techniques, i.e., quantization and hardware-oriented neural architecture search. A reduction of the bitwidths of the weights and activations of CapsNets by quantization not only lightens the memory storage requirements but has a significant impact on the energy consumption of the computational units, as demonstrated in Fig. 3.2. However, a too low numerical precision implies a decrease in accuracy, which is typically an undesired outcome from the end-user perspective. To find an efficient trade-off between the memory footprint, the energy consumption, and the classification accuracy, we propose a novel specialized framework, *Q-CapsNets*, which explores different layer-wise and operation-wise arithmetic precisions applying post-training quantization to obtaine the quantized version of a given CapsNet, with a specific focus on the dynamic routing process.

As presented in Chapter 1, neural architecture search is a method recently employed to discover novel DNN architectures in an automatized way. Targeting specifically the exploration of new capsule network models, we develop a neural architecture search framework named NASCaps. This framework not only incorporates the most common types of layers used in DNNs, such as convolutional and fully-connected, but also different types of capsule layers. Given the limitation that capsule networks pose with their computational complexity, NASCaps also takes into account different hardware efficiency parameters, such as memory usage, energy consumption, and latency, that are crucial for embedded DNN inference accelerators. Moreover, the structure of the framework is very flexible and allows us to integrate new objectives into the search easily. As a proof of concept, we propose an extended version of the NASCaps framework that searches for CapsNet models robust to

adversarial attacks, a critical aspect that has gained more and more importance recently.

The rest of the chapter is organized as follows. Section 3.2 presents the necessary background and the related works, Section 3.3 describes the framework Q-CapsNets for capsule networks quantization, Section 3.4 and Section 3.5 show the details of the NASCaps framework and its modification to support robustness to adversarial attacks.

3.2 Background and Related Works

3.2.1 Capsule Networks

Capsule networks were first introduced by Hinton et al. [5] in 2017. The two main characteristics differentiating capsule networks from CNNs are (1) the replacement of neurons with a scalar value as output with capsules returning a vector and (2) the replacement of pooling operation with a *dynamic routing* process. An output vector of a capsule is related to an entity rather than to a feature as in CNNs, and its length, i.e., its Euclidean Norm, is the instantiation probability of this entity, while the individual elements of the vector encode different spatial information, like width, skew, or rotation. Dynamic routing, or routing by agreement, makes capsule networks equivariant by modeling and learning the part-whole hierarchical relationships between entities.

The architecture of the CapsNet proposed in [5], hereon referred to as Shallow-Caps, is reported in Fig. 3.3. Since we focus on the CapsNet inference, we do not discuss the layers and algorithms only involved in the training process (e.g., decoder and reconstruction loss).

ShallowCaps is composed of the following three layers:

1. **(L1) Conv Layer:** 9x9 convolutional layer with 256 output channels;
2. **(L2) PrimaryCaps:** convolutional layer with 256 output channels. These channels are divided into 32 8-dimensional (8D) capsules (32 8D vectors of neurons). The *squash* nonlinear function forces the length of the capsule's vector to be in the range of [0:1].

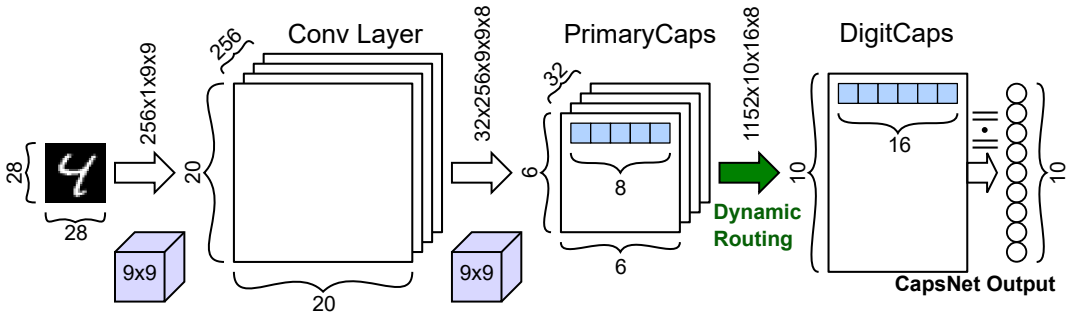


Fig. 3.3 ShallowCaps architecture for MNIST/Fashion-MNIST dataset.

3. **(L3) DigitCaps:** fully-connected layer with 16D output capsules. The number of capsules depends on the number of classes of the dataset (e.g., 10 for MNIST and FashionMNIST). Between PrimaryCaps and DigitCaps, the so-called *dynamic routing* algorithm is applied, as shown in Fig. 3.4.

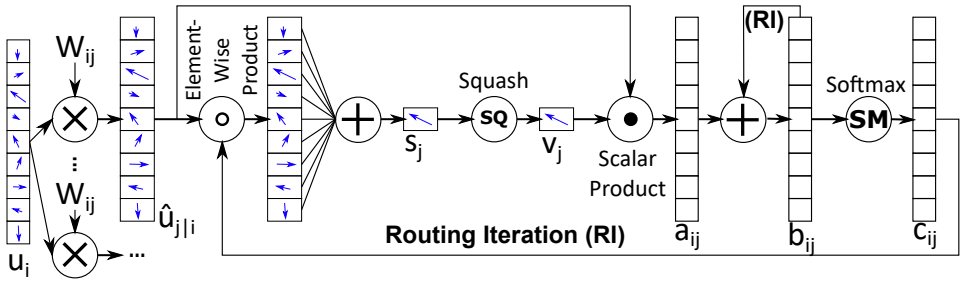


Fig. 3.4 The operations to be computed for the dynamic routing.

Dynamic routing (see Fig. 3.4) is an iterative algorithm that measures the agreement between capsules in a lower layer. Each capsule is assigned a routing coefficient. If many capsules point in the same direction with high intensity (length), they all get a high coefficient. Hence, a capsule j in a higher layer is connected to all the capsules i in the lower layer that mostly agree with each other. The computations are the following:

1. Votes $\hat{u}_{j|i} = W_{ij} \times u_i$
2. Logits initialization $b_{ij} = 0$
3. Coupling coefficients $$c_{ij} = \text{softmax}(b_{ij}) = \frac{e^{b_{ij}}}{\sum_k e^{b_{ik}}} \quad (3.1)$$
4. Preactivation $s_j = \sum_i c_{ij} \hat{u}_{j|i}$

5. Activation $v_j = \text{squash}(s_j) = \frac{||s_j||^2}{1+||s_j||^2} \frac{s_j}{||s_j||}$ (3.2)
6. Agreement $a_{ij} = v_j \cdot \hat{u}_{j|i}$
7. Logits update $b_{ij} = b_{ij} + a_{ij}$

The dynamic routing consists of iterating steps 3-7 for a defined number of times (e.g., 3 iterations in [5]).

Recently, a novel deep CapsNet architecture, *DeepCaps* [2], has been proposed (see Fig. 3.5). It introduces convolutional layers of capsules (*ConvCaps*). After the first convolutional layer with the ReLU activation function, the network features 12 ConvCaps layers. Every three sequential ConvCaps layers have an additional ConvCaps layer that operates in parallel as a skip connection. The last parallel ConvCaps layer performs dynamic routing, while the other ConvCaps layers apply the squash function. The output layer of the DeepCaps architecture is a fully-connected capsule layer with dynamic routing.

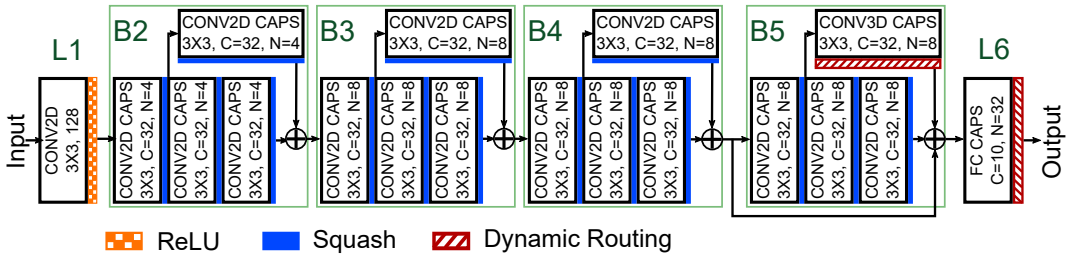


Fig. 3.5 DeepCaps architecture.

Despite the promising early results obtained by capsule networks, not many novel capsule-based architectures have been proposed in the literature [183, 184]. Most research has focused on presenting variations of the dynamic routing algorithm [185–188] and applying the models to real-world use cases, as extensively discussed in [183].

3.2.2 Quantization

As described in Section 1.2.3, a floating-point number can be converted to a fixed-point representation applying Eq. (3.3), where n is the number of bits used for the fixed-point value, and $s = \frac{x_{max}}{2^{n-1}}$ is a scaling factor with x_{max} as the largest representable

value. The advantages of fixed-point over floating-point values have been discussed in detail in the previous sections and will not be furtherly examined.

$$x_{\texttt{int}n} = \lfloor \frac{x}{s} \rceil_{\texttt{int}n} = \textbf{clamp}\left(\textbf{round}\left(\frac{x}{s}\right), -2^{n-1}, 2^{n-1}-1\right) \tag{3.3}$$

The *rounding operator* determines how the discarded fractional part influences the quantized result. Truncation (TRN) is the simplest method, as it just removes the fractional part after scaling by s. Hereon, we will denote this fractional part as ε. Round to nereast even (RTNE) sets the rule for approximating those values falling exactly halfway between two representable numbers. In particular, these numbers are rounded to the nearest even integer. Stochastic rounding (SR) operates as in Eq. (3.4).

$$\begin{cases} \lfloor x/s \rfloor & \text{if } P \geq \frac{x/s - \lfloor x/s \rfloor}{\varepsilon} \\ \lfloor x/s + \frac{\varepsilon}{2} \rfloor & \text{if } P < \frac{x/s - \lfloor x/s \rfloor}{\varepsilon} \end{cases} \tag{3.4}$$

In Eq. (3.4), $P \in [0,1)$ is a random number with uniform distribution and determines the probability of a number being rounded up or down. Fig. 3.6 shows the behavior of these three rounding operators graphically.

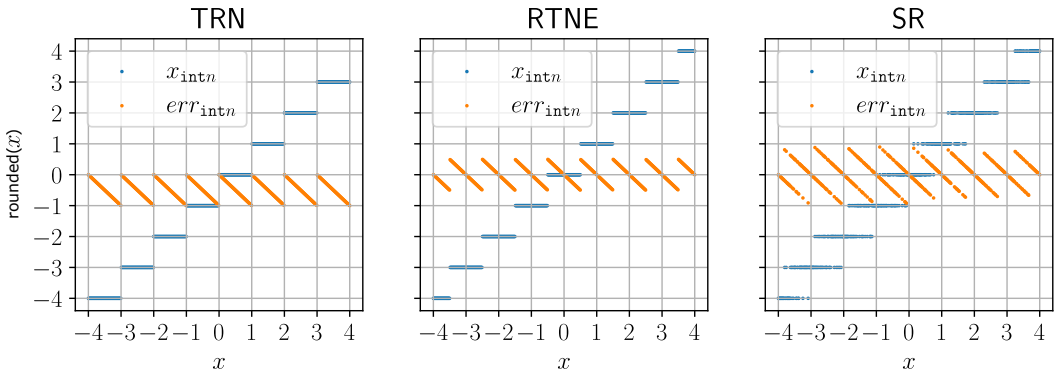


Fig. 3.6 Comparison between truncation, round-to-nearest-even and stochastic rounding operators.

A crucial characteristic of a rounding operator is the average quantization error, or *bias*, it introduces, with the quantization error computed as $err_{\texttt{int}n} = s \cdot x_{\texttt{int}n} - x$. The error of the three discussed methods is also shown in Fig. 3.6, from which we see that truncation introduces a negative bias, while RTNE and SR are unbiased.

Between RTNE and SR, the latter introduces a higher absolute error and is the more demanding from the hardware perspective as it requires generating random numbers.

Capsule Networks Quantization

At the time of the Q-CapsNets framework for capsule networks quantization development, no other works had been published on the subject. Recently, Costa et al. [189] showed how to run capsule networks at the edge on Arm Cortex-M and RISC-V MCUs. The inference is run with `int8` values, thus, the authors also perform post-training quantization from `float32` to `int8` format. Differently from our work, they do not explore the possibility of quantizing capsule networks in a fine-grained layer-wise fashion and do not focus on furtherly reducing the precision of the activations involved in the dynamic routing process, which is beneficial when designing custom hardware modules.

3.2.3 Adversarial Attacks and Robust-NAS

Despite their excellent performances, CNNs have serious security problems because adversarial attacks can trick them with slight input disturbances [190]. Studies [191, 192] have demonstrated that CNNs can be deceived by cleverly constructed inputs, and their output can be entirely altered by the addition of very minor, barely detectable perturbations to the data. To avoid being detected, the attacker must limit the added adversarial perturbation.

Formally, given a CNN model $m()$ and an input x with target classification label c, the generation of an adversarial example x^* can be formulated as a constrained optimization problem [192]:

$$\begin{aligned} x^* = \arg\min_{x^*} \quad & \mathscr{D}(x, x^*), \\ s.t. \quad & m(x) = c,\ m(x^*) = c^*,\ c \neq c^* \end{aligned} \tag{3.5}$$

where $\mathscr{D}$ is the distance between two images and the optimization objective is to minimize the adversarial perturbation to make it undetectable. x^* is considered as an adversarial example if and only if the perturbation is bounded ($\mathscr{D}(x, x^*) < \varepsilon$, where $\varepsilon \geqslant 0$) and $m(x) \neq m(x^*)$.

Goodfellow et al. [193] introduced the fast gradient sign method (FGSM) to generate adversarial examples, using the gradient of the model with respect to the input images in the highest loss direction. Madry et al. [194] and Kurakin et al. [195] presented two versions of the projected gradient descent (PGD) attack, an iterative implementation of the FGSM that introduces a perturbation α to multiple smaller steps. The PGD keeps the perturbation size minimal by projecting the generated image onto a sphere with radius ε after each iteration. It is a white-box attack with both targeted and untargeted versions. An iteration of the algorithm is the following:

$$x_i^* = x_{i-1}^* - proj_\varepsilon(\alpha \cdot sign(\nabla_x loss(\theta, x, t))) \tag{3.6}$$

Another research direction is the search for defense methods against adversarial attacks, i.e., adversarial deferences.

Recent works propose to increase the CNNs robustness against adversarial attacks using approximate computing [196] and hash-based deep compression [197], both of which come with a large hardware design overhead. Additionally, NAS approaches have been presented in recent publications to achieve high robustness against adversarial attacks. In [198], a *supernet* including all of the search space's potential architectures is trained. Subnetworks are then sampled from the supernet and assessed for accuracy and resistance to adversarial attacks. In [199], the search space is extended to include specific layer combinations that have been shown to be exceptionally strong defenses against adversarial attacks. However, the hardware efficiency issues have not yet been taken into account as conjoint optimization objectives in any of the research that focuses on NAS for adversarial attacks. Additionally, these works do not support CapsNets because they are primarily targeting CNN models.

Several surveys [190, 192] provide further details on other types of adversarial attacks and defenses.

3.3 Q-CapsNets: Framework for Capsule Networks Quantization

As discussed in the introduction to this chapter, applying quantization to capsule networks is particularly beneficial given their high computational cost and memory footprint. With this motivation, we introduce a specialized quantization framework, named Q-CapsNets, for systematically quantizing CapsNets, given a certain memory budget and accuracy reduction tolerance. Moreover, since an expensive part of CapsNets is the dynamic routing process, we add a special focus on the search for the minimal numerical precision of the activations involved in its computations. The main intent of the proposed framework is to support specialized hardware accelerator designers in having accurate estimates of (1) the minimum memory required for the weights of capsule networks and (2) the required bitwidth for custom hardware units for non-linear functions such as *squash* and *softmax*.

In the rest of the chapter, we describe the details of the proposed framework (Section 3.3.1, Section 3.3.2) and provide the results of its application to two networks [5, 2] for three datasets [4, 200, 3] (Section 3.3.3).

3.3.1 Framework Overview

The proposed Q-CapsNets framework aims to quantize CapsNets models post-training in a fine-grained fashion, where the numerical precision of weights and activations is progressively reduced until finding a trade-off between network accuracy and bitwidth, and consequently memory occupation. To speed up the process, the search space can be pruned by applying different heuristics. The framework first applies traditional techniques that can also be applied to conventional CNNs, then puts a special focus on operations characteristic of CapsNets, i.e., squash and softmax operations in the dynamic routing.

The framework (Fig. 3.7) takes as inputs:

- The *CapsNet architecture* to be quantized, together with the training and testing datasets and the necessary hyperparameters, e.g., learning rate value, schedule, and optimizer. If the network has been previously trained, it is possible to provide the pre-trained weights only.

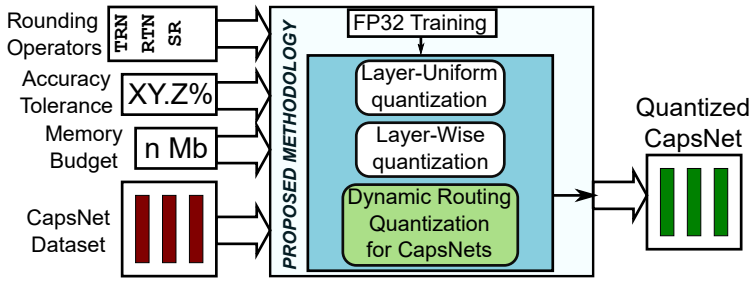


Fig. 3.7 High-level overview of Q-CapsNets framework

- A *library of rounding operators* that can be adopted to quantize the data. It is possible to provide single or multiple rounding operators. In the latter case, the framework will select the operator yielding the best result. It has to be noted that the same rounding operator will be used for all the layers of the network.
- An *accuracy tolerance* acc_{TOL}, i.e., the accepted tolerance on the accuracy loss. Considering that lowering the numerical precision is a lossy operation, the user can set a margin for quantizing the network, with which the minimum target accuracy is computed as:

$$acc_{target} = acc_{FP32} \cdot (1 - acc_{TOL}) \tag{3.7}$$

- A *maximum memory budget* available to store the weights and biases of the network.

The goal of the framework is to satisfy both the accuracy and memory footprint requirements. For the latter, the weights and biases must be quantized, aggressively if the memory budget is low. Once the parameters of the network have been quantized, there may still be some margin on the acceptable accuracy loss that allows also the reduction of the precision of the activations, to improve the energy efficiency during inference. In this situation, the framework can satisfy both accuracy and memory constraints and returns a `model_satisfied`. After quantizing the weights to satisfy the memory budget, it may happen that the overall accuracy drops below the tolerated margin. In this case, being impossible to satisfy all the requirements, the framework returns two sub-optimal solutions:

1. `model_accuracy`: a quantized CapsNet model that satisfies the target accuracy and with the minimum possible memory occupation, which is, however, higher than the one requested by the user.

2. `model_memory`: a quantized CapsNet model respecting the memory budget and achieving the highest possible accuracy, which is, however, lower than the one required by the user.

These two models can then be used as an insight to run a second iteration with more realistic constraints for the network.

3.3.2 Q-CapsNets step-by-step description

In the attempt to find a `model_satisfied`, or alternately two models `model_accuracy` and `model_memory`, the framework we propose performs several steps, as shown in Fig. 3.8. In this section, we will explain these steps at a high level. The specific implementation of each of the stages is flexible. We then propose two different versions of the framework in which some of the steps adopt different approaches, the first based on the findings of [84] and the second on the analysis of [85]. Therefore, the implementation details of these steps will be provided in the next two subsections.

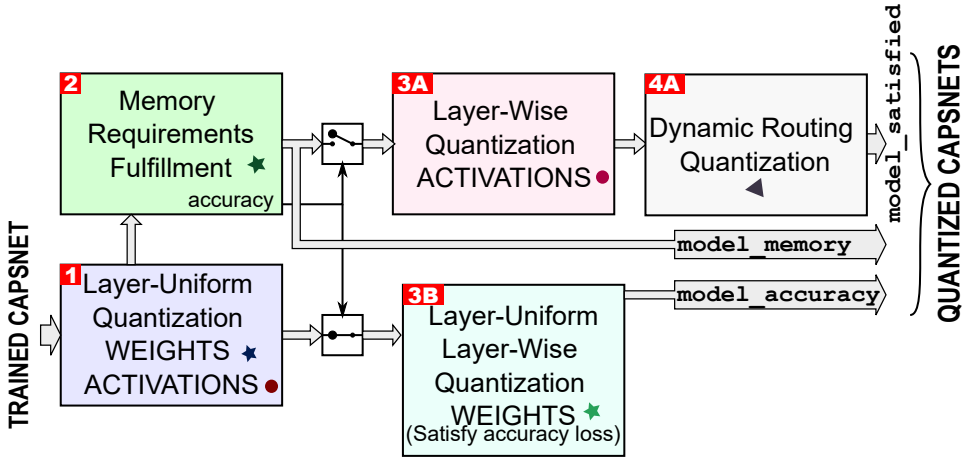


Fig. 3.8 Flow of Q-CapsNets framework for quantizing capsule networks.

As a preliminary stage, not shown in Fig. 3.8, the CapsNet provided in input to the framework is trained in full precision, i.e., FP32 format, if necessary. The resulting accuracy acc_{FP32} is used together with the input accuracy tolerance acc_{TOL},

to compute the target accuracy acc_{target} as in Eq. (3.7). Then, the steps followed by the framework (see Fig. 3.8 and Alg. 1) are the following:

1 **Layer-uniform quantization (weights+activations)**: first, all the weights and activations are converted to fixed-point representation, with the methodology explained in Section 3.2.2, using the same number of bits $Q = 32$ across the entire network. Then, the precision of both weights and activations is uniformly reduced across the network, i.e., $Q_w = Q_a$ for all the layers. The quantization in this stage can only consume 5% of the available acc_{TOL}. To speed up the search of the minimum number of bits satisfying this constraint, we adopt a binary search algorithm (lines 5-6). Both versions of the framework execute this stage in the same way.

2 **Memory requirements fullfillment**: in this step, we focus only on the quantization of the weights to fulfill the memory budget requirement. Given the memory budget and the number of weights in the model, we compute the quantization bits for the weights of each layer with the function WeightsCmptFnc (line 9). This function depends on the heuristic we decide to apply and will be further discussed in the subsections dedicated to the implementations of the two proposed framework versions. The model with these bitwidths for the weights fulfills the memory requirements and is therefore denoted as `model_memory`. Its accuracy is acc_{mm} (line 10), and we compare it with the target accuracy acc_{target}. From the result of this comparison, we decide which direction to take. If acc_{mm} is higher than acc_{target}, it means we still have a margin to quantize the activations and continue on branch **3A**. Otherwise, we jump to branch **3B**.

3A **Layer-wise quantization of activations**: in this stage, we quantize the activations of the model with a different number of bits for each layer, starting from the number of bits $Q_{a,s1}$ set in step 1 (line 7). As for the quantization of the weights, the methodology used for the activations (LayerWiseFcn, line 14) depends on the chosen heuristic and will be furtherly discussed in the next paragraphs.

4A **Dynamic routing quantization**: as described in Section 3.1, squash and softmax are computationally intensive functions, that to be executed efficiently require the design of dedicated hardware. To minimize the cost, in terms of

Algorithm 1 : Pseudo-Code of Q-CapsNets

Require: CapsNet, acc_{TOL}, memory_budget

▷ Full Precision training
model, $acc_{FP32} \leftarrow$ train(CapsNet)
$acc_{target} = acc_{FP32}(1 - acc_{TOL})$

▷ *Step 1)*
$acc_{step1} = acc_{FP32}(1 - acc_{TOL} \cdot 0.05)$
model, Q $\leftarrow$ BinarySearch(model, (model.weights, model.act), $Q_{init} = 32$, $acc_{min} = acc_{step1}$)
$Q_{w,s1} = Q_{a,s1} =$ [Q for ℓ in model.num_layers]

▷ *Step 2)*
$Q_{w,mm} =$ WeightsCmptFnc(model.weights, memory_budget)
model, $acc_{mm} \leftarrow$ test(quant(model, weights_quant=$Q_{w,mm}$, act_quant=$Q_{a,s1}$))
model_memory = model

if $acc_{mm} > acc_{target}$ **then**
    ▷ *Step 3A)*
    model, $Q_a \leftarrow$ LayerWiseFnc(model, model.act, $Q_{init} = Q_{a,s1}$, $acc_{min} = acc_{target} + 0.5(acc_{mm} - acc_{target})$)
    ▷ *Step4A)*
    $Q_{adr} = [\,]$
    for ℓ in model.num_DR_layers **do**
        Q_{adr}.append($Q_a[\ell]$)
        model_satisfied, $(Q_{adr})[\ell] \leftarrow$ DRquantFcn(model, model.DRact[ℓ], $Q_{init} = Q_{adr}[\ell]$, $acc_{min} = acc_{target}$)
    end for
    return model_satisfied

else
    ▷ *Step3B)*
    model, $Q_w \leftarrow$ BinarySearch(model, model.weights, $Q_{init} = Q_{w,s1}$, $acc_{min} = acc_{target}$)
    model_accuracy, $Q_w \leftarrow$ LayerWiseFcn(model, model.weights, $Q_{init} = Q_w$, $acc_{min} = acc_{target}$
    return model_memory, model_accuracy
end if

area and power, of the extra hardware, it is crucial to reduce the bitwidth of the activations involved in these computations. To tackle this problem, this step focuses on quantizing only the data on which squash and softmax will be applied (see Fig. 3.9).

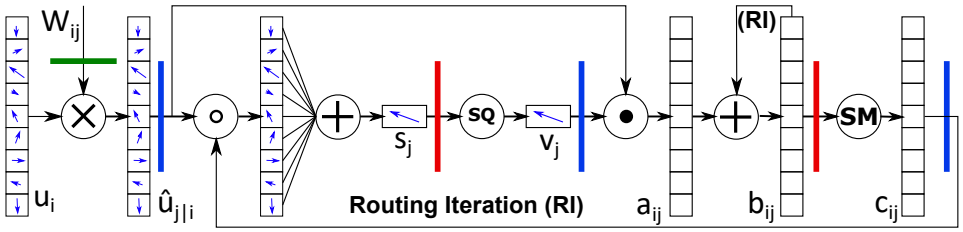


Fig. 3.9 Quantization of a capsule layer with dynamic routing. Colored bars show the tensors that are quantized. In green, the weights are quantized with Q_w bits. In blue, the activations are quantized with Q_a bits. In red, data are quantized more aggressively with Q_{DR} bits. The precision is lowered before complex and compute-intensive functions (squash, softmax).

Since the number of layers with dynamic routing in CapsNets is usually low, it is possible to do a layer-by-layer finetuning of the bitwidths of the activations involved in the dynamic routing (lines 17-20). In line 19, DRQuantFcn scales down the bitwidth of the dynamic routing activations linearly, until reaching the minimum possible accuracy.

As discussed in Section 3.2.1, dynamic routing is an iterative voting process where the strength of the votes, i.e., their magnitude, increases with the progression of iterations. As shown in Fig. 3.10, the numerical distributions of the activations on which softmax and squash are applied change significantly between iterations. For this reason, it is crucial to do loop unrolling of the dynamic routing and to define and set the proper scaling factor in the quantization process of different iterations.

3B **Layer-uniform and layer-wise quantization of weights**: this step is executed if the framework is not able to satisfy both the memory and accuracy requirements. Therefore, it aims to return a model satisfying the accuracy but with the lowest memory footprint possible. Thus, we reduce the number of bits of the weights only, first uniformly across the layers, then in a layer-wise fashion as in step 3A, until reaching acc_{target}. The resulting model is labeled `model_accuracy` and returned as the framework output together with `model_memory` generated at step 2.

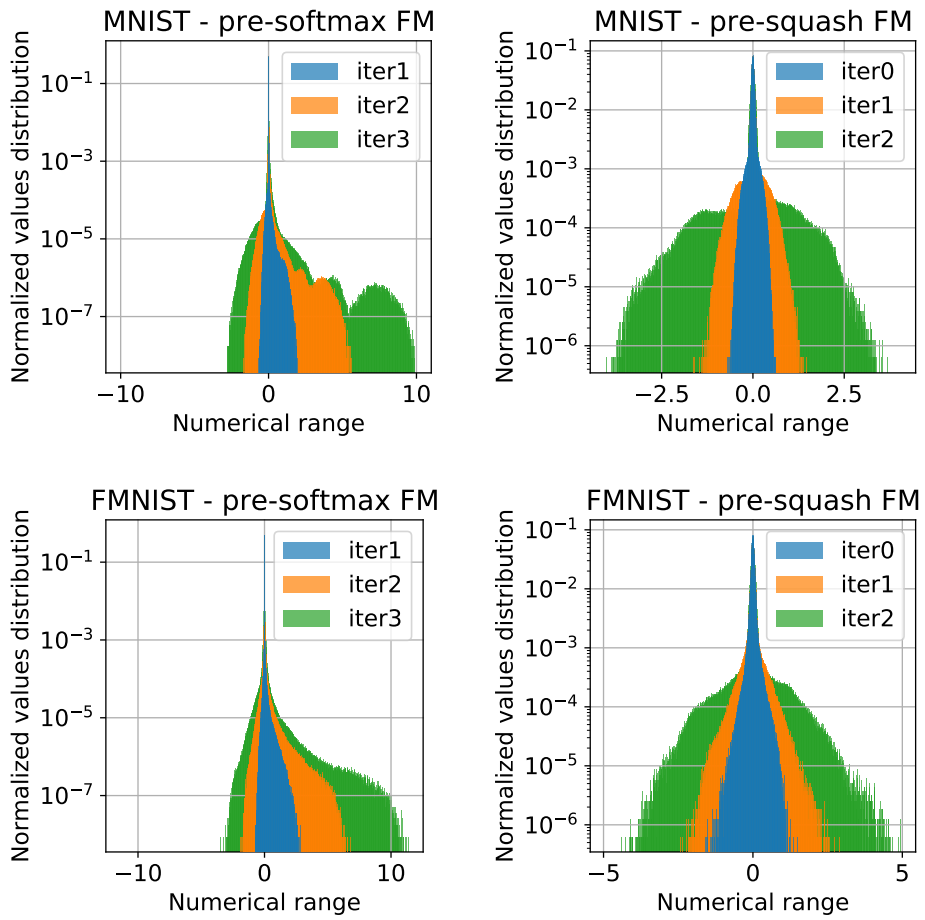


Fig. 3.10 Numerical distribution of the activations on which softmax and squash functions are applied, at different iterations of the dynamic routing algorithm.

Q-CapsNets-v1 details

As explained in the previous paragraphs, the functions to quantize in a layer-wise fashion the weights and the activations can be implemented following different heuristics. For v1 of the framework, we follow the idea of Raghu et al. [84]. According to [84], the quantization error introduced in the final layers can be higher than in the earlier layers, as the error will propagate through fewer layers and be less amplified. Based on this intuition, we implement the functions to determine the number of bits per layer so that the final layers of the network use fewer bits than the earlier layers, creating a staircase pattern.

Specifically, the number of bits for the weights is determined statically by knowing the memory budget and the number of weights per layer. For each layer ℓ we set the number of bits for the weights to $Q_w[\ell]$ so that $Q_w[\ell] = Q_w[\ell-1]-1$. With this condition set, we can compute Q_w for each layer as the maximum integer that satisfies Eq. (3.8), where L is the total number of layers and $P[\ell]$ is the number of weights of layer ℓ:

$$\sum_{\ell=0}^{L-1} (P[\ell] \cdot (Q_w[0]-\ell)) \leq \text{memory\_budget} \tag{3.8}$$

To quantize the activations, having no specific constraints, we try to reduce the bitwidth across all layers as much as possible by proceeding as shown in Alg. 2. We start from an initial number of bits Q_a, obtained, for example, in step 1. Then we select all the layers of the network except the first one and lower Q_a for all the selected layers until possible, i.e., until the accuracy remains higher than the minimum accuracy of the stage acc_{min}. Then, we fix the bitwidth used for the activations of the first two layers and repeat the Q_a reduction step for all the remaining layers until all the bitwidths are fixed.

Algorithm 2 : Algorithm for Layer-wise Activations Quantization - v1

1: Given: Q_{init} initial number of quantization bits to start the algorithm, acc_{min} minimum value of accuracy that can be reached.
Require: model, Q_{init}, acc_{min}
2: L = model.num_layers
3: Q = [Q_{init} for ℓ in model.num_layers]
4: $start\_\ell = 1$
5: **while** $start\_\ell$ <L **do**
6: $acc = \text{inf}$
7: **while** $acc \geq acc_{min}$ **do**
8: $Q[\ell]$ -= 1 for ℓ in [$start\_\ell$, ..., L]
9: model, $acc \leftarrow$ test(quant(model, act_quant=Q))
10: **end while**
11: $Q[\ell]$ += 1 for ℓ in [$start\_\ell$, ..., L]
12: $start\_\ell$ += 1
13: **end while**
14: **return** quant(model, params $\leftarrow Q$), Q

Q-CapsNets-v2 details

For v2 of the framework, we consider the analysis of [85], where the effect of the quantization on the accuracy of neural networks is measured with the signal-to-quantization noise ratio (SQNR). The key finding in [85] is that "*all the quantization steps contribute equally to the overall SQNR of the output, regardless if it's the quantization of weights, activations, or input, and irrespective of where it happens (at the top or bottom of the network)*".

In step 2, to compute the bitwidth of the weights in each layer satisfying the memory budget, we first characterize the weights in terms of SQNR, then sort the layers from highest to lowest SQNR as shown in Fig. 3.11. We divide the sorted layers into three groups and compute the total number of weights P_g for each group. Then, we set as a constraint to use more bits for the low-SQNR layers, and fewer bits for the high-SQNR layers, as in Eq. (3.9). With Eq. (3.9), we determine the number of bits Q_w that satisfies the memory budget.

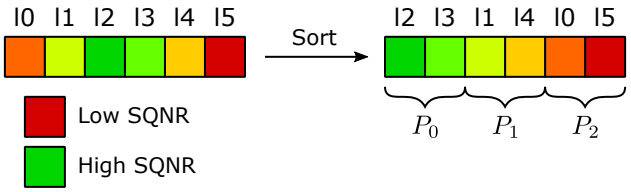


Fig. 3.11 Layers sorting by weights SQNR and grouping.

$$\sum_{g=0}^{3} (P_g \cdot (Q_w + g)) \leq \text{memory\_budget} \tag{3.9}$$

The model obtained after this operation is denoted as `model_memory`. If, after this step, the model still satisfies the accuracy requirement, the framework proceeds directly to step 3A to quantize the activations furtherly. If, on the contrary, the accuracy is too low, Q_w is increased until the accuracy requirement is met again (and the memory budget is not). Then, iteratively, one layer is popped from the SQNR-sorted list and its weights bitwidth $Q_w[\ell]$ is lowered until the accuracy requirement is not met anymore. The iterations on the sorted list stop if and when the memory budget is satisfied. In this case, the framework proceeds with step 3A. If, on the contrary, the algorithm iterates through all the layers and, in the end, the memory budget is not satisfied, it means that the framework can not satisfy both accuracy and memory requirements, and it jumps to step 3B.

Algorithm 3 : Algorithm for Layer-wise Activations Quantization - v2

Require: model, memory_budget, acc_{min}
$Q_w \leftarrow$ Eq. (3.9)
model, $acc \leftarrow$ test(quant(model, weight_quant=Q_w))
if $acc \geq acc_{min}$ **then**
    return model, Q_w
else
    while $acc < acc_{min}$ **do**
        Q_w += 1
        model, $acc \leftarrow$ test(quant(model, weight_quant=Q_w))
    end while
    Q_w = [Q_w for i in model.num_layers]
    SQNR_list = [(ℓ, ℓ.SQNR) for ℓ in model.layers]
    sort_by_SQNR(SQNR_list)
    for ℓ in SQNR_list **do**
        while $acc > acc_{min}$ **do**
            $Q_w[\ell]$ -= 1
            model, $acc \leftarrow$ test(quant(model, weight_quant=Q_w))
        end while
        $Q_w[\ell]$ += 1
        if Eq. (3.9)[Q_w] is satisfied **then**
            return model, Q_w, take_branch_A
        end if
    end for
    return model, Q_w, take_branch_B
end if

For what concerns the activations quantization, a similar method is applied (Alg. 3). As explained for v1 of the framework, there is no constraint on the activations bitwidth. Therefore, the aim is to reduce it as much as possible. In a very fine-grained way, we could apply the same algorithm used for the weights, i.e., sort the layers by activations SQNR and tune the bitwidth layer-by-layer. However, having no constraints, the process is necessarily carried out for all the layers, requiring a very long time. To increase the speed of the framework, we reduce the granularity of the activations quantization by sorting the layers by activations SQNR and dividing them into 4 groups. The bitwidths of the activations of all the layers within a group are then tuned together, i.e., the activations will have the same number of bits. The number of groups used is a hyperparameter of the framework and can be set to trade-off between speed and granularity.

Rounding Operator Selection

If multiple rounding operators are provided in input, a quantized model is generated for each of them. Note that for different rounding operators, the framework could take different paths, i.e., path A or B, because of varying rounding errors. Depending on whether the algorithm followed Path A or not, the best rounding operator is selected using the following criteria.

A) **There are some models generated from Path A:**

1) Models from Path B are discarded.
2) The model with lower memory is selected.
3) With the same memory, the model with fewer bits used to represent activations is selected.
4) With the same memory and bits for the activations, the model with the simplest rounding operator is selected, e.g., with our examples, in order, truncation, round-to-nearest-even, and stochastic rounding. Note, while the first one simply requires the deletion of the LSBs, the last one requires more complex operations to decide the orientation of the rounding.

B) **There are models only from Path B:**

1) In this case, two models are returned. Selecting from `memory_model`, the model with the highest-possible accuracy is returned.
2) Selecting from `accuracy_model`, the model with the lowest-possible memory is returned.
3) If more than one model has the same highest accuracy and the lowest memory occupancy, the simplest rounding operator is preferred to break the tie.

3.3.3 Results

Experimental Setup

We implement the Q-CapsNets framework (see Fig. 3.12) in PyTorch [40], and we run it on two Nvidia GTX 1080 Ti GPUs. We test it on the ShallowCaps model [5],

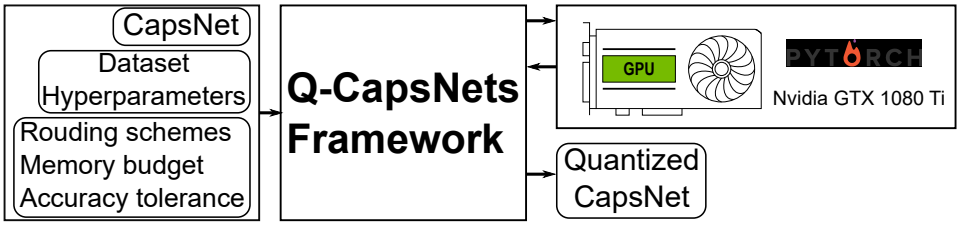


Fig. 3.12 *Experimental setup to test our Q-CapsNet framework.*

on MNIST [4] and FashionMNIST [200] datasets, and on the DeepCaps model [2] on the MNIST, FashionMNIST and CIFAR10 [3] datasets. The MNIST database is a collection of 28x28 grayscale handwritten digits, from 0 to 9, with 60,000 training samples and 10,000 testing samples. The FashionMNIST is a collection of 28x28 grayscale images representing Zalando's articles associated with 10 classes. It is composed of 60,000 training samples and 10,000 testing samples. The CIFAR10 is a collection of 32x32 color images organized into 10 different classes, with the training set composed of 50,000 samples and the testing set of 10,000 samples. For full precision training, data augmentation is applied as follows:

- MNIST: images are randomly shifted by a maximum of two pixels and rotated by 2 degrees;
- FashionMNIST: images are randomly shifted of 2 pixels and horizontally flipped with a probability of 0.2;
- CIFAR10: images are resized to 64x64[1], randomly shifted of 5 pixels, rotated of 2 degrees and horizontally flipped with a probability of 0.5.

No data augmentation is applied to the test set images.

Q-CapsNets-v1 Results

All the results discussed in this section have been obtained with the RTNE rounding method, which, as shown later, is the best-performing one among the tested rounding operators.

For the MNIST dataset, the ShallowCaps architecture [5] is trained in full precision (FP32), for 100 epochs and with a batch size equal to 100. We use an exponential

[1]The original images of size 32x32 are resized to 64x64 by bilinear interpolation, to allow deeper networks, as reported in the original paper [2].

decay learning policy, with an initial learning rate of 0.001, 2000 decay steps, and a 0.96 decay rate. Its achieved test accuracy is 99.67%.

Since the framework has a conditional path, for clarity, we present two examples corresponding to the execution of the different branches of the algorithm.

Test of the Path A: For the first set of experiments, we test the Path A of the framework, i.e. when both the memory and accuracy constraints are satisfied. Since the memory requirement at FP32 is 25.6MB, we set an accuracy tolerance of 0.2% and a memory reduction of 5×, i.e., a memory budget of 5.12MB. The result in Fig. 3.13 [Q1] shows that the `model_satisfied` reduces the memory footprint of the weights by 5.53× compared to the FP32 model, with an accuracy equal to 98.71%, thus satisfying all the requirements. Along with the reduction of the memory occupied by the weights (W mem), we report the reduction of the volume of activations (A vol) as a metric to evaluate the bitwidth reduction and, consequently, the computational savings. For `model_satisfied`, this volume is reduced by 5.33×, and only 3 bits are necessary for the activations involved in the dynamic routing operations.

Test of the Path B: Since our framework executes Path B if it cannot find a solution that satisfies both memory and accuracy requirements, for its testing purpose, we provide as input a very low memory budget, requiring an 8× memory reduction and a 0.2% accuracy tolerance. The result of the experiment, shown in Fig 3.13, indicates that to satisfy the memory requirements, the weights of `model_memory` [Q2] are set to very low bitwidths, causing a too-high reduction of accuracy. To satisfy the accuracy requirements in `memory_accuracy` [Q3], weights are reduced to the minimum possible bitwidth. From 3.13, we can also see that in `model_memory` and `model_accuracy` no optimizations are performed on the activations bitwidths, and appreciate the descending staircase behavior determined by the chosen heuristic.

Similar sequences of tests are performed on the same ShallowCaps model for the FashionMNIST dataset, whose floating-point accuracy is 92.79%. Two illustrative results from our experiments are reported in Tab. 3.2.

As for the ShallowCaps architecture, several tests are run on the DeepCaps model for the MNIST, FashionMNIST, and CIFAR10 datasets. Fig. 3.14 reports graphically some key results obtained with the Q-CapsNets framework on the DeepCaps for the CIFAR10 dataset, where [Q4] identifies a `model_satisfied` solution, while [Q5] and [Q6] the `model_memory` and `model_accuracy` solutions respectively. It has

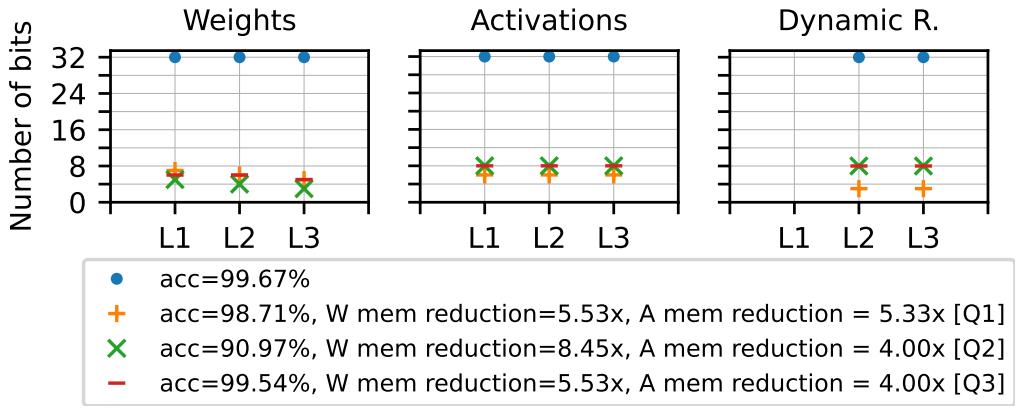


Fig. 3.13 Q-CapsNets-v1 example results of the ShallowCaps for the MNIST dataset.

Table 3.2 Q-CapsNet-v1 accuracy results, weight (W) memory and activation (A) volume reduction for the ShallowCaps and for the DeepCaps on MNIST, Fashion-MNIST, and CIFAR10 datasets.

Model	**Dataset**	**Accuracy**	**W mem red.**	**A vol red.**	**DR bits**
ShallowCaps	MNIST	99.31%	6.69x	4.57x	(6, 4)
ShallowCaps	MNIST	98.71%	5.33x	5.33x	(3, 3)
ShallowCaps	FMNIST	92.65%	3.27x	3.56x	(8, 6)
ShallowCaps	FMNIST	91.98%	4.11x	4.00x	(6, 6)
DeepCaps	MNIST	99.47%	10.74x	4.93x	(4, 4)
DeepCaps	MNIST	99.65%	4.01x	4.93x	(3, 3)
DeepCaps	FMNIST	94.24%	8.04x	3.81x	(9, 4)
DeepCaps	FMNIST	94.28%	3.21x	4.29x	(3, 3)
DeepCaps	CIFAR10	91.18%	3.01x	2.61x	(11, 6)
DeepCaps	CIFAR10	90.62%	4.20x	3.28x	(7, 4)

to be noted that the layers within a CapsBlock are grouped and tuned in parallel, i.e., their activations ad weights will have the same bitwidth, reducing thus the time required for search. Therefore, in Fig. 3.14, the labels beginning with *B* refer to a CapsBlock. Some key results are reported in Tab. 3.2 as well.

QCapsNets-V2 Results

The same set of tests described for v1 of the framework has also been conducted with v2. In particular, in Fig. 3.15, we see the Path A [Q7]/Path B [Q8][Q9] test on the ShallowCaps model for the MNIST dataset. Interestingly, we see that the bitwidths of the weights still follow a descending staircase pattern, as in v1 of the framework.

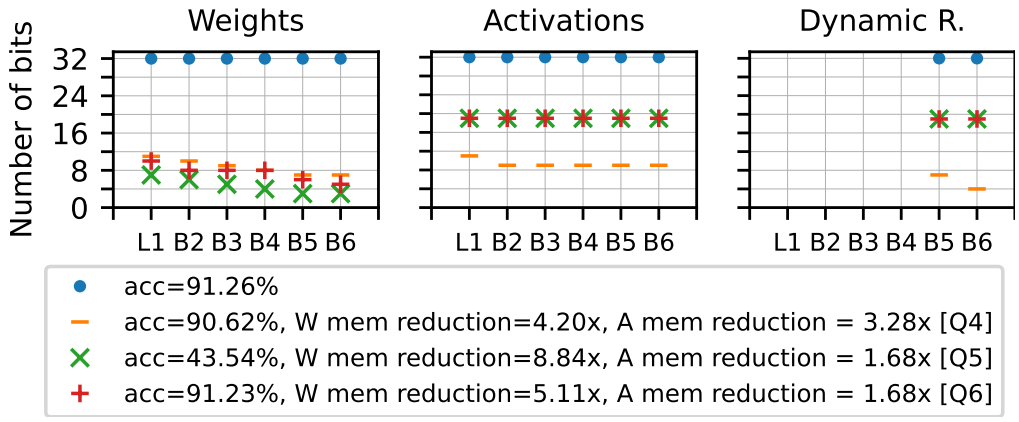


Fig. 3.14 Q-CapsNet-v1 example results of the DeepCaps for the CIFAR10 dataset.

This is due to the values of the SQNRs of the weights that show this behavior for this specific model. On the contrary, on the activations, we see an opposite trend. In v1 as in v2, the bitwidths of the activations of `model_memory` and `model_accuracy` are not furtherly tuned after step 1 if Path B is followed.

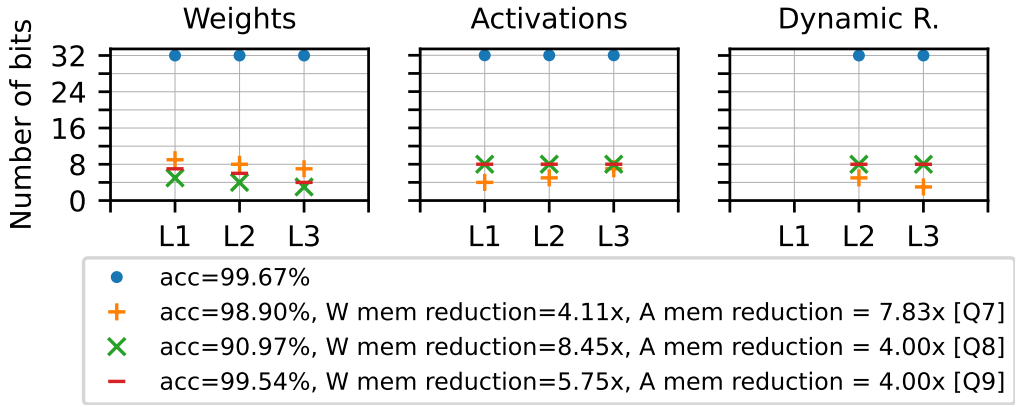


Fig. 3.15 Q-CapsNets-v2 example results of the ShallowCaps for the MNIST dataset.

The framework's difference between v1 and v2 is more visible in Fig. 3.16 since the DeepCaps architecture, here tested on the CIFAR10 dataset, has more layers. Here we see that both the weights and activations bitwidths don't follow a specific pattern since the extent to which they are reduced depends on the SQNR of each layer. Some key results for all the combinations of networks and datasets are reported in Tab. 3.3.

To evaluate the better-performing version between the two proposed, we conduct the same set of tests with both v1 and v2, swiping on three accuracy tolerance constraints and eight memory reduction requirements. Fig. 3.17 shows how the two

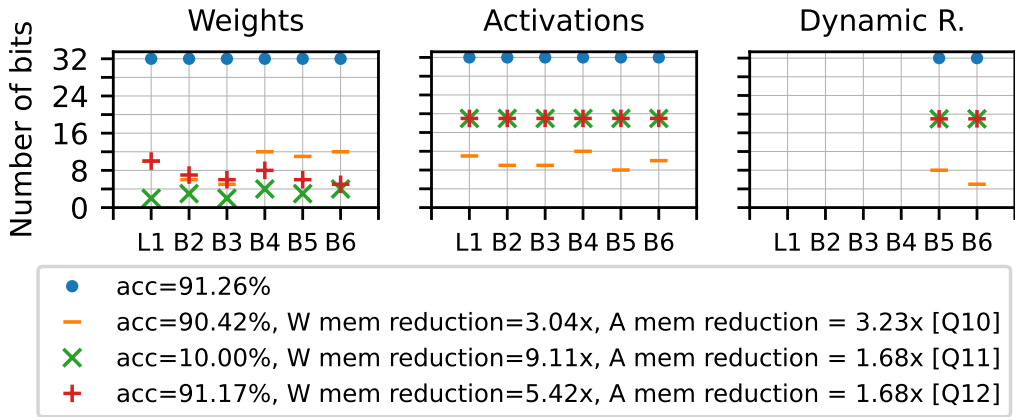


Fig. 3.16 Q-CapsNets-v2 example results of the DeepCaps [2] for the CIFAR10 [3] dataset.

Table 3.3 Q-CapsNets-v2 accuracy results, weight (W) memory and activation (A) volume reduction for the ShallowCaps and for the DeepCaps on MNIST [4], Fashion-MNIST and CIFAR10 datasets.

Model	**Dataset**	**Accuracy**	**W mem red.**	**A vol red.**	**DR bits**
ShallowCaps	MNIST	98.74%	7.99x	4.52x	(5, 5)
ShallowCaps	MNIST	99.20%	3.27x	7.68x	(5, 3)
ShallowCaps	FMNIST	92.65%	4.57x	4.41x	(6, 6)
ShallowCaps	FMNIST	91.88%	7.20x	3.92x	(11, 8)
DeepCaps	MNIST	99.35%	11.20x	3.29x	(2, 3)
DeepCaps	MNIST	99.67%	5.25x	4.83x	(2, 5)
DeepCaps	FMNIST	94.24%	3.24x	4.51x	(3, 3)
DeepCaps	FMNIST	94.35%	10.73x	3.75x	(5, 4)
DeepCaps	CIFAR10	90.49%	2.06x	3.33x	(8, 4)
DeepCaps	CIFAR10	91.13%	5.42x	1.97x	(11, 6)

versions perform on all the combinations of networks and datasets. Specifically, each box corresponds to a test with certain accuracy tolerance and memory budget constraints, and it is marked if the framework can return a `model_satisfied` for these requirements. The orange triangle is the marker for v1, and the purple dot is for v2. The dashed lines help visualize the most stringent constraints that each framework version can satisfy, and we easily see that v2 always outperforms v1. The motivation behind this result can be found in many works, such as [201, 202], that exhaustively analyze the effect of the quantization of each individual layer on the overall accuracy of the network. The outcome is that the first and last layers are often the worst offenders. While the SQNR analysis (v2) intrinsically takes

into account this characteristic, v1 of the framework blindly applies a staircase quantization pattern with which the last layers are the most strongly quantized.

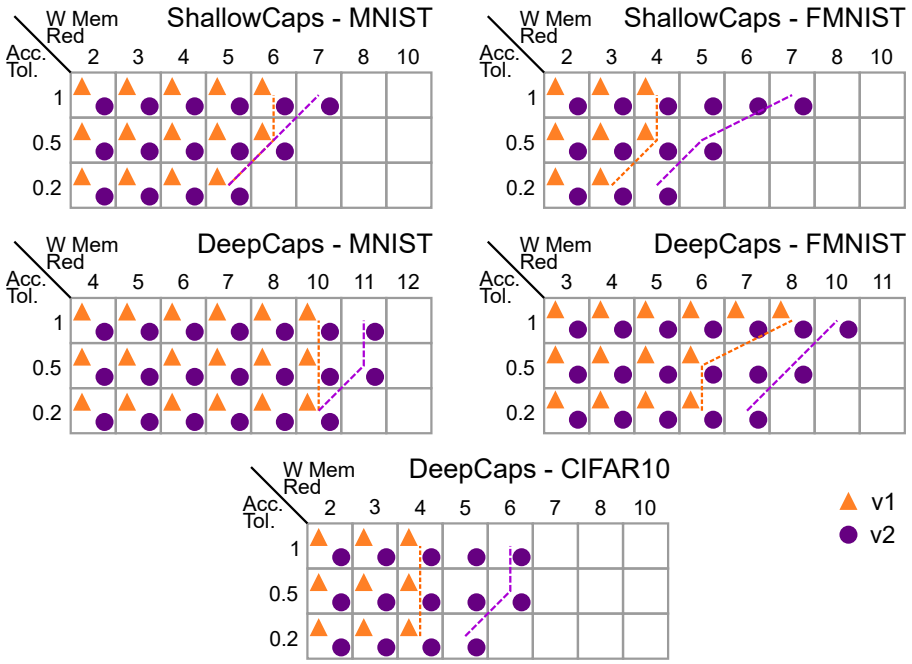


Fig. 3.17 Comparison between v1 and v2 of Q-CapsNets framework. Each box corresponds to a test with certain accuracy tolerance and memory budget constraints. The box is marked only if the framework is able to return a `model_satisfied` for that test.

Rounding Operators Comparison

Fig. 3.18 shows the results of comparing the three rounding operators described in Section 3.2.2 for all combinations of networks and datasets. As for the comparison between v1 and v2 of the framework, each box identifies a test with specific accuracy tolerance and memory budget constraints. All the tests are performed three times, each time providing a single rounding operator as input of the framework. A box is marked if the framework can return a `model_satisfied` for the corresponding test. The figure shows that truncation is the worst-performing method in all situations, as expected, given the bias it introduces. On the contrary, RTNE and SR have comparable performances, with RTNE still slightly outperforming SR, as anticipated by the higher absolute error that SR introduces. RTNE is the preferable method for the results it achieves and its simple hardware implementation.

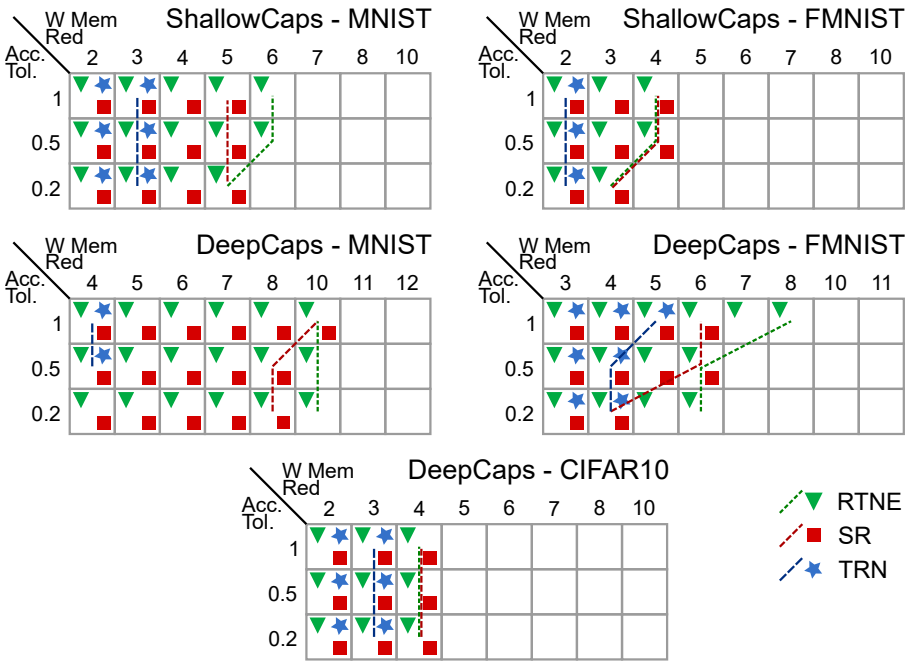


Fig. 3.18 Comparison between TRN, RTNE, and SR rounding methods. Each box corresponds to a test with certain accuracy tolerance and memory budget constraints. The box is marked only if the framework can return a `model_satisfied` for that test.

3.3.4 Conclusions

With the Q-CapsNets framework we demonstrate how with quantization it is possible to greatly reduce the numerical precision used for both the weights and activations of capsule networks by switching from a `float` format to an `int` format. With the proposed approach, the memory footprint of the two studied models can be reduced up to 11×, depending on the specific network-dataset combination. Moreover, we pay particular attention to the activations involved in the squash and softmax operations of dynamic routing, managing to reduce their bitwidth down to 3 bits. Such extreme reduction is a great benefit in case one wants to integrate custom units for these operations into a dedicated accelerator. The purpose of this framework is to be a fast tool to get a lower-bound estimate of how much capsule network models can be quantized, with quantization applied post-training without any fine-tuning. As a future development, one can further push the limits of quantization of capsule networks by applying more advanced, though slower, techniques, such as precisely fine-tuning post-quantization or quantization-aware training.

3.4 Neural Architecture Search for Hardware Efficient Capsule Networks

NAS is a widely adopted technique to automatize the exploration of new DNN models, mostly focusing on their accuracy [203, 34]. Recently, given the popularity of specialized hardware accelerators, evaluations on hardware efficiency have been introduced in NAS algorithms [204–207], thus expressing the quality of a DNN not only in terms of accuracy but also of memory footprint or required FLOPS. To the best of our knowledge, none of them include the possibility of employing capsule layers and dynamic routing in the design space, which are fundamental blocks of CapsNets.

To fill this gap, we present NASCaps, a framework for the NAS of DNNs that not only includes the most popular DNN layer types (such as convolutional and fully-connected) but also, for the first time, the various capsule layer types. Our framework is multi-objective since it evaluates network accuracy and several hardware-efficiency metrics essential for embedded DNN inference accelerators, such as memory utilization, energy consumption, and latency. However, the search space can be extremely vast because of the many possible configurations. In the time required to explore and evaluate candidate networks, one must consider the time required for DNNs training, which can be very long [208], and for the precise post-synthesis hardware measurements. To overcome these challenges, we use a genetic algorithm to efficiently explore the entire search space (Section 3.4.4). Moreover, we avoid full training of networks by proposing a technique based on Pearson's correlation coefficient [209], with which we estimate the accuracy of partially trained networks (Section 3.4.5). Finally, we show how to model the execution of the candidate networks given an accelerator architecture, making precise post-synthesis measurements unnecessary (Section 3.4.3).

3.4.1 NASCaps Overview

By performing a multi-objective NAS to find a set of accurate yet resource-efficient DNN models, our multi-objective NASCaps framework generates and evaluates convolutional- and capsule-based DNNs. This is done by jointly taking into account the DNN validation accuracy, energy consumption, latency, and memory footprint.

The search is based on a specialization of the genetic NSGA-II [210] algorithm to enable a multi-objective comparison and selection among the generated candidate DNNs.

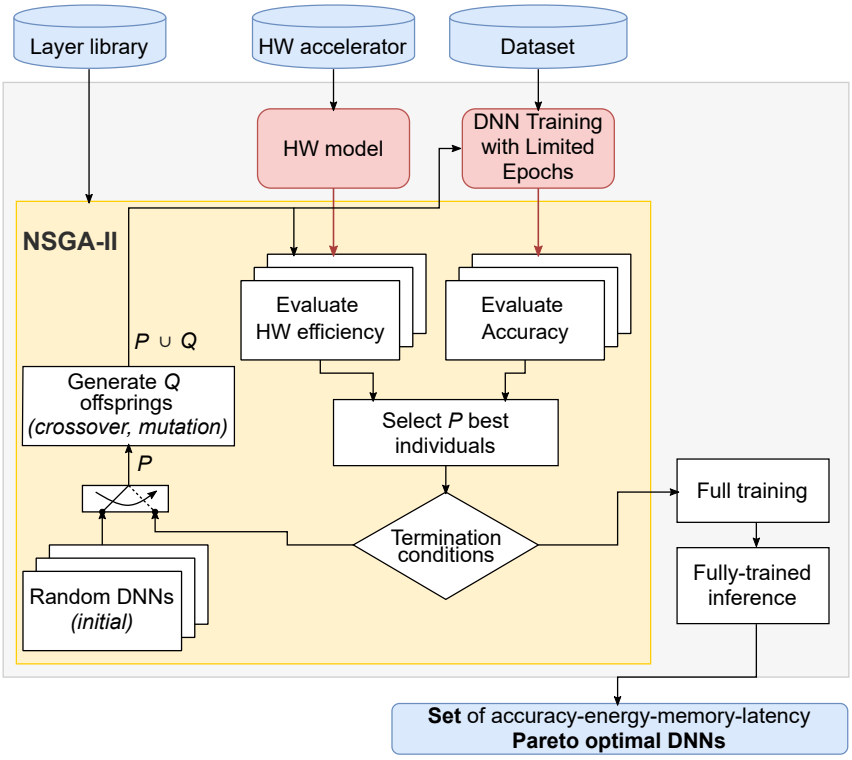


Fig. 3.19 Overview of our *NASCaps* framework, showing different components and their interconnections defining the workflow.

Fig. 3.19 depicts the general structure and workflow of the NASCaps framework. It receives in input a set of potential types of layers that can be used to create various candidate DNNs, as well as the configuration of the underlying hardware accelerator, which would execute the resulting DNN in a real-world scenario. Convolutional layers, capsule layers (as described in [5]), and the CapsCell and FlatCaps layers described in [2] are all included in the layer library we first constructed. Due to the modular design of our framework, we anticipate that future versions will be able to easily incorporate additional types of layers to broaden the search space. This is also possible because we use a straightforward modular representation of the candidate networks that relies on combining single-layer descriptors, as discussed in Section 3.4.2.

N randomly generated DNNs are used as the input for the automated search to start the evolutionary process. At each iteration of the evolutionary process, the candidate DNNs are only partially trained. As we will see in Section 3.4.5, this optimization is intended to lower the computational cost and time necessary for the search while maintaining a high level of correlation to the full-training accuracy, as evaluated by the Pearson correlation coefficient. The partially-trained DNNs are then characterized in terms of accuracy, memory footprint, energy consumption, and latency. After the evaluation stage, the evolutionary algorithm determines a new Pareto-frontier, and the top potential DNN solutions are used as inputs for the next iteration.

At the end of the evolutive process, the Pareto-optimal DNN solutions are fully trained[2] to assess their accuracy precisely. We go into full detail on the core aspects of our system in the following subsections.

3.4.2 Parametric Modeling of Capsule-Based Layers and Networks

For the compact description of candidate DNNs architectures, the NasCaps framework relies on an explicit position-based representation. Each layer is defined by a *layer descriptor*, and a whole network is built by stacking these descriptors. Each descriptor encodes all the information required to uniquely describe a layer in a concise form, with a 9-element position-based structure. The elements of the layers descriptor are:

(1) type of layer

(2) size of the input feature maps n_{in}

(3) number of input channels ch_{in}

(4) size of input capsules $caps_{in}$

(5) kernel size $kernel_{size}$

(6) stride size $stride_{size}$

[2] A complete training up to the 100th epoch for the MNIST, Fashion-MNIST, and SVHN datasets, and up to the 300th epoch for the CIFAR-10 dataset is conducted.

(7) size of the output feature maps n_{out}

(8) number of output channels ch_{out}

(9) size of output capsules $caps_{out}$

This representation is modular, enables the description of any candidate DNN, and is flexible, as it allows the description of complex structures by simply defining a new *layer type*. For example, a repeating structure such as the CapsBlock of the DeepCaps architecture can be defined with a single layer descriptor. In this way, the whole DeepCaps architecture is described with six descriptors, one for the first convolutional layer, four for the CapsBlocks, and one for the final capsule layer.

A DNN architecture is described by stacking the proper layer descriptors. It is completed by a *skip-connection* field to encode the position of skip connections if present and a *resize flag* explicitly indicating the need for input resizing. Fig. 3.20 shows the format of the proposed layer descriptor and the description format of a generic DNN architecture, from now on referred to as *genotype*.

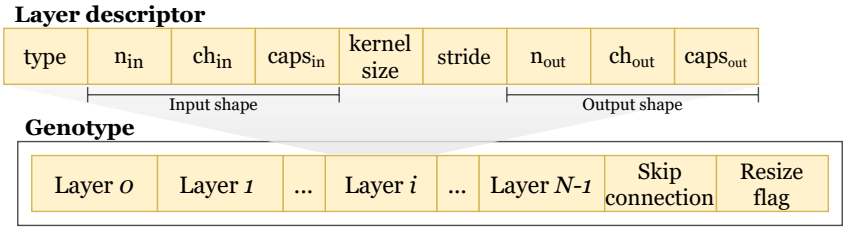


Fig. 3.20 Proposed structure of the genotype.

3.4.3 Modeling of CapsNets Execution on Hardware Accelerators

One of the NASCaps framework inputs is the hardware accelerator structure targeted for the inference of the DNN. To model the execution of any candidate DNN on the input accelerator, it is first necessary to extract the *layer-specific parameters* for each supported layer type. Second, from the RTL-level description of the accelerator, we have to extract and model the micro-architectural configuration at a higher abstraction level, condensing the information in some *global parameters*, which will be used in the evolutive process.

For illustration, we showcase the modeling of the CapsAcc [182] accelerator, as it supports the execution of capsule layers. The array of processing elements (PEs) that constitute the CapsAcc computing core is followed by an accumulator that properly adds the partial sums. The accumulator output is then processed by an activation unit that may apply ReLU, softmax, or squash functions. Three buffers are employed throughout computation to maximize data reuse and decrease access to larger memories. The activations and weights are stored in data and weight memories, respectively, and the partial outputs of the dynamic routing iterations are stored in a routing buffer. A control unit determines the paths for the mapping of various layers onto the PE array.

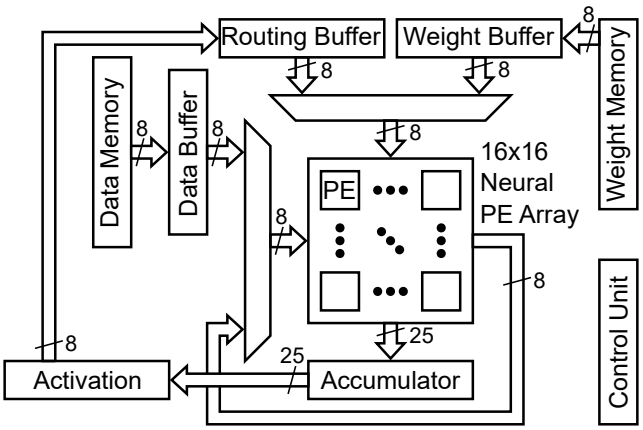


Fig. 3.21 Architectural view of the CapsAcc accelerator.

Layer-specific modeling

The needed layer-specific parameters to be extracted for each layer are the following:

- *weights*: the number of weights of the layer,
- *sums_per_out*: number of terms to be accumulated to compute one output value,
- *data_per_weight*: number of feature maps multiplied by the same weight.

The equations used to compute these layer-specific parameters depend on the layer type, as shown in Tab. 3.4. It has to be noted that if the parameters $caps_{in}$ and $caps_{out}$ are set to 1, the *ConvCaps* and *ClassCaps* layers become standard Conv and fully-connected layers, respectively.

Table 3.4 Equations for the operation-specific modeling of CapsNets.

Operation	*weights*	*sums_per_out*	*data_per_weight*
ConvCaps layer	$(ch_{in} \cdot kernel_{size}^2 + 1) \cdot ch_{out} \cdot caps_{out} \cdot caps_{in}$	$(kernel_{size}^2 + 1) \cdot ch_{in} \cdot caps_{in}$	$(n_{out})^2 \cdot ch_{in} \cdot caps_{in}$
ConvCaps3D layer	$(ch_{in} \cdot kernel_{size}^3 + 1) \cdot ch_{out} \cdot caps_{out} \cdot caps_{in}$	$(kernel_{size}^3 + 1) \cdot ch_{in} \cdot caps_{in}$	$(n_{out})^2 \cdot ch_{in} \cdot caps_{in}$
ClassCaps layer	$(ch_{in} \cdot n_{in}^2 + 1) \cdot ch_{out} \cdot caps_{out} \cdot caps_{in}$	$(n_{in}^2 + 1) \cdot ch_{in} \cdot caps_{in}$	1
Dynamic Routing	$ch_{in} \cdot kernel_{size}^2 \cdot ch_{out}$	$caps_{in}$	1

Global parameter modeling

The modeling needs to estimate the memory footprint, latency, and energy consumption of the inference of a CapsNet on the given hardware accelerator. While the memory footprint can simply be computed as the sum of the number of weights of each layer, the estimation of the latency and energy consumption needs micro-architectural information on the hardware being used. In the case of the CapsAcc, considering the available bandwidth, hardware resources, and a weight stationary dataflow, the modeling happens as follows.

Regarding dataflow, the weights are first loaded on-chip in the PEs array registers, where they are reused for all the needed iterations. This loop is repeated until all the output feature maps of a layer are computed for all the layers.

The CapsAcc architectural parameters are the following:

- w_load_cycles: the number of cycles to load a group of weights on-chip, equal to 16 given the PEs array dimension,
- en_{mem}: the energy consumption of single memory access,
- pwr_{PEA}: the average power consumption of the PEs array.

Given these architectural parameters, the number of groups of weights loaded on-chip for each layer is computed as:

$$w\_loads = \left\lceil \frac{weights}{16 \cdot \min(16, sums\_per\_out)} \right\rceil \tag{3.10}$$

and the number of cycles per layer as:

$$cycles(l) = w\_load\_cycles \cdot w\_loads + data\_per\_weight \tag{3.11}$$

The overall latency of the whole DNN is the sum of the contributions of all the layers:

$$latency = \sum_{l \in L} cycles(l) \cdot T \tag{3.12}$$

where T is the clock period.

To compute the number of memory accesses *ma*, it is necessary to distinguish whether the layer is convolutional. As shown in Eq. (3.13), this can be done by analyzing the value of $data\_per\_weight$, which is higher than 1 in the case of Conv layers.

$$ma = \begin{cases} 256, & \text{if } data\_per\_weight = 1 \\ 16 \cdot \max(sums\_per\_out - 15, 1), & \text{otherwise} \end{cases} \tag{3.13}$$

The energy consumption of the accelerator is then estimated considering both the memory accesses and PEs array contributions:

$$energy = \left\lceil \frac{ma \cdot 8}{128} \right\rceil \cdot en_{mem} + \sum_{l \in L} cycles(l) \cdot T \cdot pwr_{PEA} \tag{3.14}$$

3.4.4 The Multi-Objective NSGA-II Algorithm

The core of the NASCaps framework for selecting the Pareto-optimal solutions is based on the NSGA-II evolutionary algorithm [210], reported in Alg. 4. The main loop (lined 2-14) represents a single generation of the complete evolution process of an initial population P_1. The initial population P_1 of N solutions is randomly generated (line 1). This set is the initial parent generation and the first iteration's input.

For each iteration, mutations and crossover among solutions of the parent population P_g generate a new set of offspring solutions Q_g (line 3). The whole population $P_g \cup Q_g$ is evaluated (line 4) and sorted according to a non-domination criterion to select the N best individuals that will populate P_{g+1} (lines 5-15). Iteratively, the solutions of the whole population are grouped in different Pareto-fronts F_i, where F_1 represents the best-found solutions, while the subsequent fronts F_i are constructed by removing from the population the solutions belonging to the preceding fronts $F_{1,\ldots,i-1}$. The extraction of Pareto-solutions goes on until having filled the next

Algorithm 4 : The genetic NSGA-II algorithm used in our *NASCaps* framework.

Require: search space S, sizes of population $|P|, |Q|$, number of generations G

1: $P_1 \leftarrow RandomConfigurations(|P|)$
2: **for** $g = 1 \dots G$ **do**
3: $\quad Q_g \leftarrow CrossoverAndMutate(P_g, |Q|)$
4: $\quad T_g \leftarrow EstimateParameters(P_g \cup Q_g)$
5: $\quad P_{g+1} \leftarrow \emptyset$
6: $\quad i = 1$
7: $\quad$ **while** $|P_{g+1}| < |P|$ **do**
8: $\quad\quad F_i = PickPareto(T_g)$
9: $\quad\quad$ **if** $|P_{g+1}| + |F_i| \leq |P|$ **then**
10: $\quad\quad\quad P_{g+1} \leftarrow P_{g+1} \cup F_i$
11: $\quad\quad$ **else**
12: $\quad\quad\quad P_{g+1} \leftarrow P_{g+1} \cup DistanceCrowding(F_i, |P| - |P_{g+1}|)$
13: $\quad\quad$ **end if**
14: $\quad\quad i+ = 1$
15: $\quad$ **end while**
16: **end for**
17: **return** $PickPareto(P_g)$

generation P_{g+1} with N solutions. To have exactly N individuals, the solutions of the last front are sorted using a crowded distance comparison approach (line 11). With this method, the solutions of the front are sorted according to each objective function in ascending order. Overall, at the end of the selection process, only half of the $P_g \cup Q_g$ population will be selected to be part of the P_{g+1} parent individuals of the next generation. The algorithm is also graphically shown in Fig. 3.22.

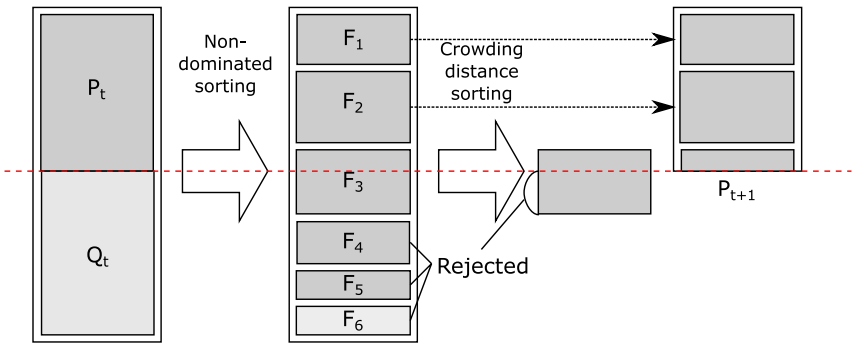


Fig. 3.22 Sorting of the population.

The steps just described are repeated for a number of g generations. Alg. 4 reports the complete pseudocode, and the following procedures are used:

- $RandomConfigurations(N)$: returns N randomly-generated configurations sampled in the search space.
- $CrossoverAndMutate(X,N)$: applies crossover and mutation (Section 3.4.4), N offsprings are generated from the parents P_g.
- $PickPareto(X)$: given a set X, the Pareto-solutions are identified and returned. These solutions are also removed from the set X.
- $DistanceCrowding(X,N)$: given a set X, sorts the solutions by applying crowded distance comparison and returns the N best solutions.

The advantage of this multi-objective genetic algorithm is that the Pareto-fronts are re-constructed at each generation, aiming to cover all the possible solutions and return a non-dominated set of solutions.

Crossover and mutation

The two operations used in the genetic algorithm evolution are crossover and mutation. Single-point crossover generates two offsprings solutions Q_a and Q_b from two parents P_a and P_b, randomly picked among all the parents solutions. The two parents' genotypes are split into two parts, where the splitting points are randomly picked. Then, solution Q_a is formed by appending the tail of the P_b genotype to the head of P_a genotype, and vice versa for Q_b. Since the splitting points are randomly chosen, it is necessary to check the validity of the two generated solutions. In particular, we verify that all solutions have at least one initial convolutional layer and two capsule layers and that no standard convolutional layer is inserted between capsule layers. The reason behind the second check is that the purpose of capsule layers is to work at a higher level of abstraction w.r.t. standard convolutional layers. Fig. 3.23 graphically shows the crossover operation.

The second key operation, i.e., mutation, is implemented by randomly choosing one of the layer descriptors from the genotype of a solution in the P set and one of the parameters in it, and randomly modifying it with probability p_m. The parameters

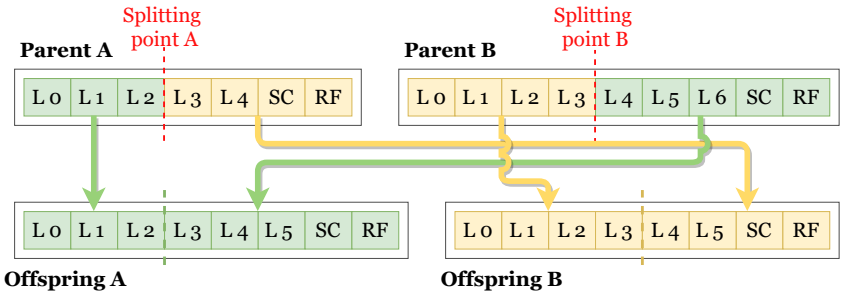


Fig. 3.23 Example of crossover between two genotypes.

that can be mutated are the stride, the kernel size, the number of output capsules, or the position of the skip connections.

After applying cross-over and mutation, a final correcting step is performed, during which offsprings solutions are fixed by properly adjusting the input and output tensors dimensions of layers if necessary.

3.4.5 Results

Experimental setup

Fig. 3.24 shows our experimental setup and flow. The candidate DNNs have been implemented with the Tensorflow library [41]. All the training and validations have been performed on GPU-HPC computing nodes equipped with four NVIDIA Tesla V100-SXM2 GPUs. The framework has been tested on the MNIST [4], FashionMNIST [200], SVHN [211], and CIFAR10 [3] datasets. The hardware metrics, i.e., energy efficiency and latency, have been evaluated using the open-source CapsAcc [182] accelerator as a reference HW platform. The core processing elements of the accelerator have been synthesized using Synopsys Design Compiler, a 45nm technological node, and a clock period $T = 3ns$.

Three different sets of experiments have been conducted. (I) A basic random search has been performed to study the number of epochs necessary to train and evaluate the candidate DNNs. (II) To find Pareto-optimal DNNs for memory, energy, latency, and accuracy, the genetic search algorithm has been executed. (III) The resulting DNNs have been fully-trained. To study the transferability of the selected DNNs model for a certain dataset, the training is also conducted on the other datasets.

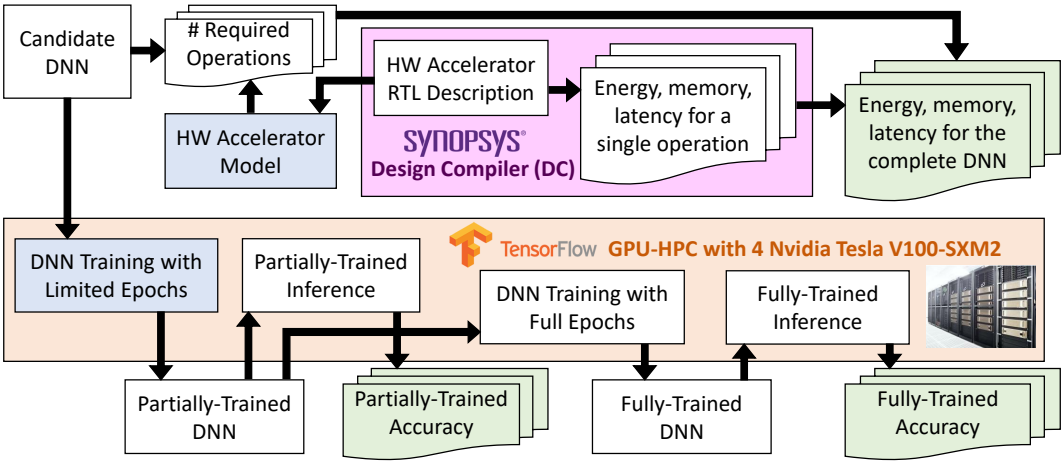


Fig. 3.24 Setup and tool-flow for conducting our experiments.

To perform the experiments, the following settings have been used:

- $|P| = 10$ as initial parent population size,
- $|Q| = 10$ as offspring population size,
- $g = 20$ as the maximum number of generations per genetic loop,
- $p_m = 10\%$ as mutation probability,
- $kernel_{size} \in \{3\times 3, 5\times 5, 9\times 9\}$
- $stride_{size} \in \{1,2\}$
- $ch_{out} \in \{1,2,\ldots,64\}$
- $caps_{out} \in \{1,2,\ldots,64\}$

These numbers, along with the training hyper-parameters (such as batch sizes, number of epochs, and learning rate), were chosen after executing a number of early experiments and taking into account realistic runtimes.

Results for Reduced Training Epochs for Full-Training Accuracy Estimation

The high computational cost of the exploration process is one of the crucial aspects of NAS. This problem is caused by the population's vast number of candidate networks

and the expensive training process required to assess their accuracy. We present a two-stage evaluation strategy to reduce the amount of time required to complete the search and, as a result, its computational cost.

The initial randomly generated DNNs and all the candidate DNNs generated during the evolutionary process are trained only for a short number of epochs, thus forming a set of DNNs that are only partially trained. The accuracy evaluation of the Pareto-fronts in the NSGA-II method, as presented in Section 3.4.4, is done on these partially-trained DNNs. The number of epochs to be used for the partial training has been carefully chosen by studying the effect of this new hyperparameter on the final achieved accuracy of the candidate networks. Only the selected candidates are fully trained and validated at the end of the evolutionary process.

This method enables the prediction of the DNNs' full-training accuracy using a smaller number of training epochs. It was evaluated utilizing 66 randomly generated DNNs (along with the ShallowCaps and DeepCaps architectures) and performing full training on each of them while also logging the obtained validation accuracy at each training epoch. The accuracy of the fully trained DNNs and the accuracy of the same DNNs at the intermediate training steps have been compared using the Person correlation coefficient (PCC) [209].

Tab. 3.5 shows the values of the PCC between the accuracy of the DNNs after n training epochs and after the full training, together with the median cumulative time needed to perform an n-epochs training. This study made it possible to establish that, as expected, more complex datasets require a higher number of training epochs to accurately identify the most promising networks among all the candidate solutions. 5 epochs are adequate in the case of the MNIST dataset to achieve a *PPC* of 0.9999. Instead, such a high confidence level is never obtained within the first few epochs for the CIFAR-10 dataset. In this case, 10 training epochs are chosen, ensuring a PCC of 0.9334. The correlation coefficient and the time needed for training are trade-offs in this decision. Naturally, more training epochs can be chosen, but doing so would significantly extend the exploration time owing to the consequently higher DNN training time — an important factor to take into account when the NASCaps framework explores huge populations and/or an elevated number of generations is used. On the other hand, the selection process carried out after 10 training epochs rather than 5 enables more Pareto-dominated candidate networks to be discarded.

Table 3.5 Pearson correlation coefficient (PCC) and median cumulative training time expressed in seconds (MCTT) for the MNIST, Fashion-MNIST (FMNIST), SVHN, and CIFAR-10 datasets.

Epoch n.		1	3	5	10	15	20
MNIST	PCC	0.8407	0.9998	0.9999	1.0000	1.0000	1.0000
	MCTT	55.4	166.2	277.0	554.0	831.0	1108.0
FMNIST	PCC	0.8306	0.8963	0.9013	0.9935	0.9989	0.9998
	MCTT	86.2	258.7	431.1	862.3	1293.4	1724.6
SVHN	PCC	0.6812	0.8733	0.9518	0.9531	0.9667	0.9876
	MCTT	128.3	385.0	641.6	1283.3	1924.9	2666.6
CIFAR-10	PCC	0.2969	0.4259	0.7279	0.9334	0.9518	0.9879
	MCTT	61.6	184.7	307.9	615.8	923.6	1231.5

NASCaps Results for Partially-Trained DNNs

The MNIST dataset is first used to test the effectiveness and proper functioning of our NASCaps framework. The number of generations is set at 20, although a maximum time-out of 12 hours has been enforced for the MNIST and Fashion-MNIST datasets and 24 hours for the CIFAR10 and SVHN datasets.

For the MNIST-NAS, the search lasted 20 complete generations, with each candidate network trained for five epochs. 210 DNNs were trained and evaluated using this technique. The selected solutions are compared to the two reference SOTA models, i.e., the ShallowCaps and DeepCaps architectures. In Fig. 3.25a, the performance of each DNN is shown in terms of accuracy, energy, memory footprint, and latency, i.e., the four objectives of the search.

The Fashion-MNIST search was completed in 19 generations (12 hours), and 200 candidate architectures were assessed. It took 12 generations to complete the search for the SVHN dataset, and 130 architectures could be evaluated. With 150 different models investigated, the search for the CIFAR-10 dataset has reached the 14^{th} generation.

The progression of the evolutionary search algorithm for the MNIST and CIFAR-10 datasets is depicted in Fig. 3.25. Notice that the red dots, i.e., the candidate models of generation 0 (see pointer ① in Fig. 3.25a), represent randomly generated DNNs. In the following iterations, when our evolutionary method discovers better candidate DNN structures by iteratively utilizing crossover and mutation operations, the objectives considerably increase (see pointer ②).

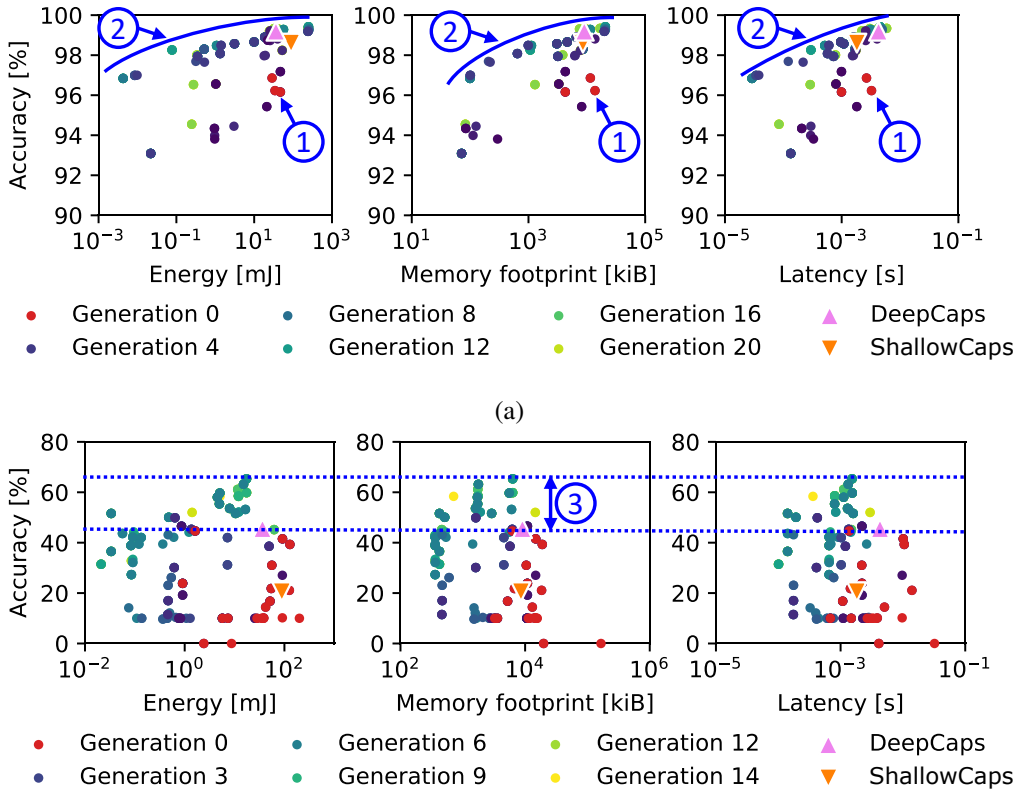


Fig. 3.25 Partially-Trained DNN NAS for (a) the MNIST dataset, and (b) the CIFAR-10 dataset. The color shows in which generation the solution occurs first.

A significant number of candidate networks (almost 700 designs) based on convolutional and capsule layers were evaluated thanks to the reduced-epoch training. This technique produced several candidate architectures with accuracy levels up to 30.86% higher than the partially-trained SOTA solutions, i.e., within the constraints of a significantly shorter training period. For instance, the DeepCaps architecture only managed to achieve a network accuracy of 45.60% during the same training time as one of the networks found by the NAS for the CIFAR-10 dataset (see pointer ③ in Fig. 3.25b). This supports the idea that, when constrained to a shorter training time, our NASCaps framework can produce networks with higher accuracy than DeepCaps-like structures.

NASCaps Results for the Selected Fully-Trained DNNs

After the evolutionary search, the candidate DNNs from the Pareto-optimal subsets have been fully trained to assess their final accuracy. The Pareto-optimal solutions at the end of the full training process are shown in Fig. 3.26.

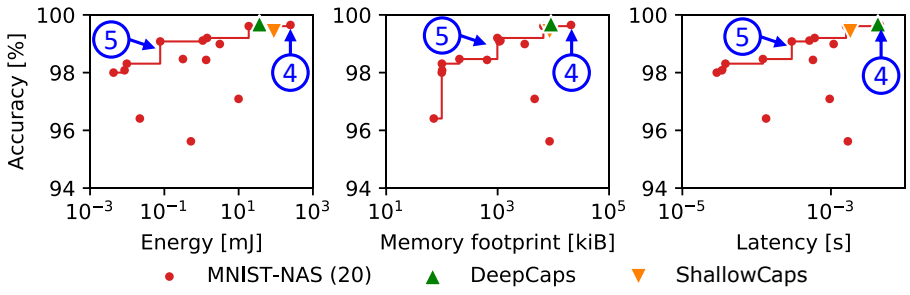
NASCaps for the MNIST Dataset. In 93 training epochs, the highest-accuracy architecture (pointer ④ in Fig. 3.26a) discovered during the MNIST search achieved an accuracy of 99.65%. However, compared to the CapsNet model, it uses 2.8× as much energy, 2.5× as much time, and 2.4× as much memory. The red front (see pointer ⑤) also draws attention to other intriguing solutions that are part of the derived Pareto-optimal front. These solutions have slightly lower accuracy but up to a couple of orders of magnitude lower energy, memory, and latency.

NASCaps for the Fashion-MNIST Dataset. In 51 epochs, one of the best solutions (pointer ⑥ in Fig. 3.26b) achieves an accuracy of 92.15%. In comparison to both the CapsNet and DeepCaps architectures, this solution reduced latency (-79.38%), energy (-88.43%), and memory footprint (-63.05%) while maintaining nearly the same accuracy (93.94%).

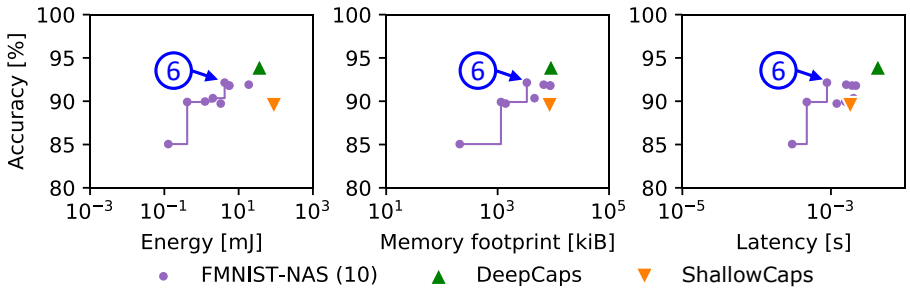
NASCaps for the SVHN Dataset. In 56 training epochs, the set of experiments for the SVHN dataset generated a solution (see pointer ⑦ in Fig. 3.26c) that achieved an accuracy of 93.17%, i.e., 3.52% less than the DeepCaps. However, compared to DeepCaps, this method drastically lowered energy by 97.05% and latency by 29.56%, but it uses 1.6× more memory. As opposed to the DeepCaps, another interesting model (see pointer ⑧) achieved a 92.53% accuracy while using 30.59% less energy, 59.63% less delay, and 62.70% less memory.

NASCaps for the CIFAR-10 Dataset. After 300 training epochs, a solution discovered by the CIFAR10-NAS (see pointer ⑨ in Fig. 3.26d) outperformed the DeepCaps architecture in all the objectives and reached an accuracy of 85.99%. In comparison to the DeepCaps run on the CapsAcc accelerator, this solution (NASCaps-C10-best in Tab. 3.6) lowered energy usage by 52.12%, latency by 64.34%, and memory footprint by 30.19%, while experiencing a modest accuracy drop of roughly 1% using the same training settings. Tab. 3.6) lists additional Pareto-optimal DNN architectures for the CIFAR-10 dataset that our NASCaps framework identified.

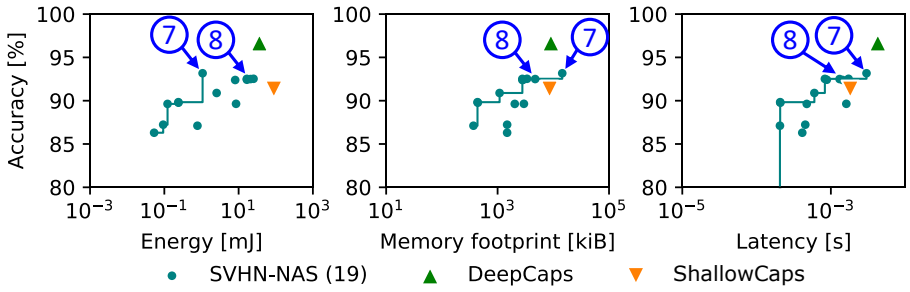
Transferability of the Selected DNNs Across Different Datasets. The dataset-specific identified DNNs have also been trained and tested on the other datasets



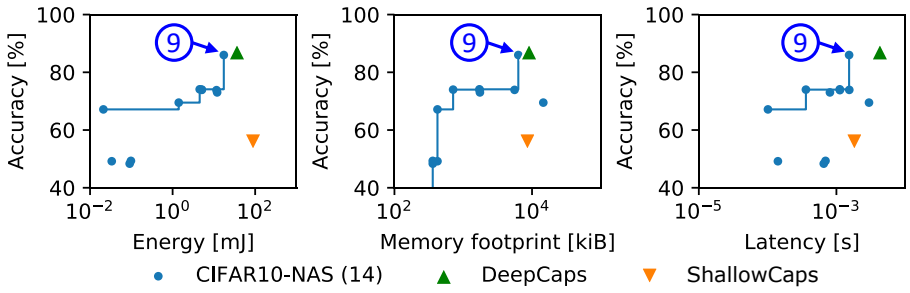
(a)



(b)



(c)



(d)

Fig. 3.26 Fully-trained DNN results for (a) the MNIST, (b) the Fashion-MNIST, (c) the SVHN, and (d) CIFAR-10 datasets.

under investigation to assess the transferability of the DNN solutions discovered by

Table 3.6 Selected CIFAR-10 architectures after 300-epoch training. Note that the accuracy reported for the DeepCaps and CapsNet do not 100% match with the ones reported in the original papers [5, 2]. This can be attributed to the differences in the training hyper-parameter setup, as their papers do not disclose the complete in-depth information about the training that can ensure the reproducibility of their results.

Architecture	Accuracy	Energy	Latency	Memory
DeepCaps [2]	87.10%	36.30 mJ	4.29 ms	9,052 kiB
NASCaps-C10-best ⑨	85.99%	17.38 mJ	1.53 ms	6,319 kiB
NASCaps-C10-a0d	74.11%	4.53 mJ	1.12 ms	1,718 kiB
NASCaps-C10-9fd	74.00%	5.11 mJ	0.36 ms	713 kiB
NASCaps-C10-658	73.91%	5.06 mJ	1.54 ms	5,573 kiB
CapsNet [5]	55.85%	88.80 mJ	1.82 ms	8,573 kiB

our NASCaps methodology. The matrix of solutions with the maximum accuracy, as determined by this transferability analysis, is reported in Tab. 3.7.

Table 3.7 Highest-Accuracy DNNs found by the dataset-specific NAS, which are then trained for the other datasets for 100 epochs.

Architecture	MNIST	FMNIST	SVHN	CIFAR-10
NASCaps-MNIST-best ④	99.65%	93.34%	96.36%	71.44%
NASCaps-FMNIST-best ⑥	99.49%	92.15%	93.12%	68.34%
NASCaps-SVHN-best ⑦	99.51%	91.43%	93.17%	63.72 %
NASCaps-C10-best ⑨	**99.72%**	**93.87%**	**96.59%**	**76.46%**

For the other datasets, the NASCaps-C10-best architecture of Tab. 3.6 turned out to be especially accurate. It outperformed the MNIST-NAS solutions with an accuracy of 99.72% for the MNIST dataset after 37 training epochs. It achieved an accuracy of 93.87% for the Fashion-MNIST dataset after 32 epochs of training, which is higher than the DeepCaps accuracy after 100 epochs. It outperformed the highest-accuracy DNN discovered during the SVNH-NAS and achieved an accuracy of 96.59% when tested on the SVHN dataset. Like DeepCaps, the NASCaps-C10-best design contains two initial convolutional layers and three CapsCell blocks but no skip connections. The MNIST-NAS highest-accuracy architecture also proved effective with the Fashion-NMIST dataset, achieving an accuracy of 93.34% after 91 training epochs.

The findings presented in Tab. 3.7 demonstrate that the NASCaps-C10 solution is the best overall architecture discovered during the four searches. This is due to several factors, including the fact that the evolutionary process was based on a randomly selected beginning parent population that was generated anew at each search. Furthermore, it's possible that the initial parent population's modest size prevented the four dataset-specific searches from converging. Also, at the conclusion of the experiments, not all four searches had arrived at the same generation.

3.4.6 Conclusions

NASCpas is a NAS framework that for the first time also explores capsule network architectures. Moreover, the framework does not only focus on accuracy but jointly searches for hardware-efficient models, potentially easing the deployment of DNNs based on capsule layers on resource-constrained IoT/edge devices. For example, one of the CapsNets discovered for the CIFAR10 dataset has an 86.0% accuracy, with an energy consumption per inference of 38.63mJ, a memory footprint of 11.85MB, and a latency of 4.47ms.

3.5 Neural Architecture Search for Robust Capsule Networks

NASCaps is a very flexible framework that easily allows the integration of new objectives in the search. As a proof of concept, we show how to integrate robustness to adversarial attacks to NASCaps. The security of DNNs classifiers has recently become a major concern since small and invisible to the naked eye perturbations can be added to the inputs [190] and completely change the classification result. This is a dangerous threat to safety-critical applications [212]. Usually, the robustness of DNNs to adversarial attacks is evaluated a posteriori once the model has already been designed. By integrating robustness to adversarial attacks in the search objectives of NASCaps, we enable a conjoint search of accurate, HW-efficient, and robust models.

3.5.1 Integration of Adversarial Robustness Evaluation in NASCaps

Integrating new objectives in the NASCaps framework is easy, as shown in Fig. 3.27. It is simply necessary to add a metric when evaluating the candidate solutions and, consequently, an objective when extracting the Pareto-fronts. In the discussed case, the new metric is the robustness to adversarial attacks.

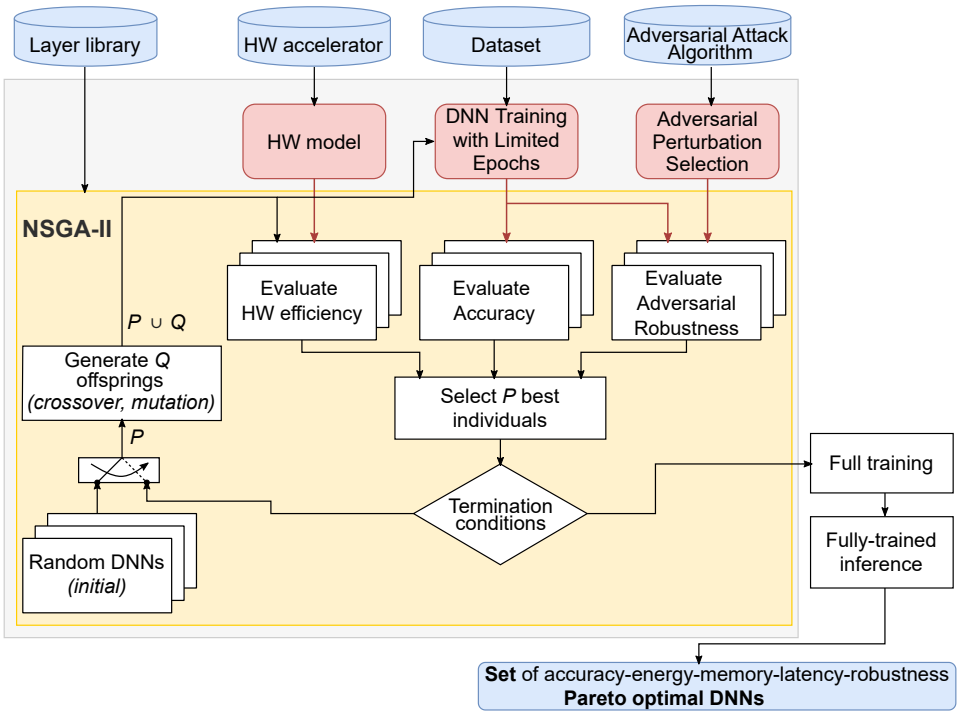


Fig. 3.27 Variation of the NasCaps framework to add robustness to adversarial attacks as a search objective

Since the design space can explode by taking into account many types and strengths of adversarial perturbations, its size is limited by automatically choosing the value of adversarial perturbations to be employed in the NAS for a particular dataset. The proposed method is summarized in Alg. 5. The PGD algorithm [194] generates an adversarial example for each item of the testing dataset (line 5). Note that additional adversarial attack algorithms may be incorporated into our system, PGD is used here for demonstrative purposes.

The amount of adversarial perturbation is determined by the parameter ε. As shown in Section 3.5.2, when looking at the variation of the accuracy w.r.t. ε, there is a region where the accuracy drop is higher, i.e., it has a steep slope. The region with the highest slope is around half of the clean accuracy, i.e., $\frac{Acc_0}{2}$. By taking advantage

of this understanding, we determine ε_{NAS} as the adversarial perturbation value that yields an accuracy as close as possible to the $\frac{Acc_0}{2}$. The *One EPS* search, which optimizes for the robustness against one value of perturbation, uses the chosen value of ε_{NAS}. The *Two EPS* search is also set up to cover a wider adversarial perturbation range. After choosing ε_{low} and ε_{high} (lines 10–11), the NAS is carried out while optimizing for the adversarial accuracy with both values.

Algorithm 5 : Adversarial Perturbation Selection.

Require: Deep Neural Network: N; Test Dataset: $\mathscr{D} = \bigcup_j X_j$;

1: Adversarial Perturbation Budget: $\varepsilon_i \in \mathscr{E} = [\varepsilon_{MIN}, \varepsilon_{MAX}]$;
2: $Acc_0 = Accuracy(N(\mathscr{D}))$
3: **for** $i \in < \mathscr{E} >$ **do**
4: **for** $j \in < \mathscr{D} >$ **do**
5: $X'_{ij} = PGD(N, \varepsilon_i, X_j)$
6: **end for**
7: $\mathscr{D}'_i = \bigcup_j X'_{ij}$
8: $Acc_i = Accuracy(N(\mathscr{D}'_i))$
9: **end for**
10: $\varepsilon_{NAS} = \varepsilon_i \ : \ Acc_i \approx \frac{Acc_0}{2}$
11: $\varepsilon_{low} \approx \frac{\varepsilon_{NAS}}{10}$
12: $\varepsilon_{high} \approx 3 \cdot \varepsilon_{NAS}$
13: **return** $\varepsilon_{NAS}, \varepsilon_{low}, \varepsilon_{high}$

3.5.2 Results

The setup used for the experiments is the same as shown in Section 3.4.5. As in the previous section, we present three sets of results: we first show the outcomes of selecting the adversarial perturbation values, then the results for the partially-trained DNNs, and finally those for the fully-trained selected DNNs.

Selection of the Adversarial Perturbation for the NAS

One important factor to be considered when running the NAS is the level of adversarial perturbation. The Pareto-optimal DNNs of the NASCaps library have been tested under the PGD attack using various values of the adversarial perturbation

ε, according to the methodology outlined in Section 3.5.1. The results shown in Fig. 3.28 demonstrate that the accuracy of DNNs decreases as ε increases. Tab. 3.8 reports the values selected for the NAS.

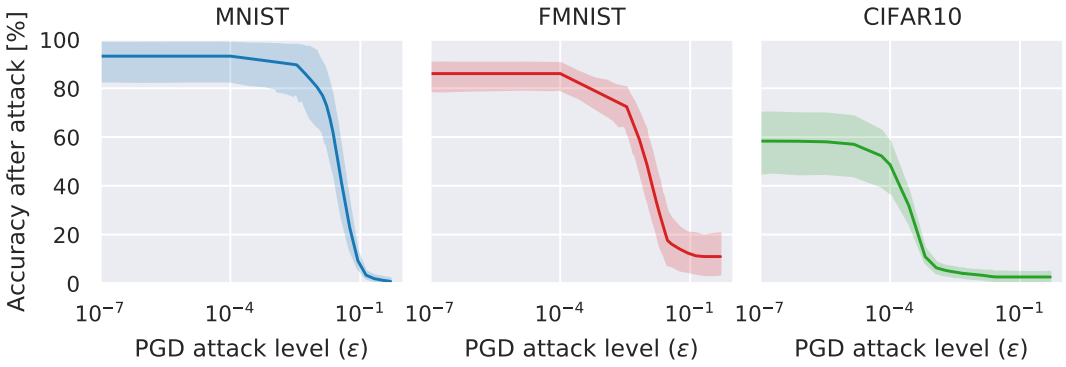


Fig. 3.28 Analysis of the DNN robustness under the PGD attack, with different adversarial perturbation values, for MNIST, Fashion-MNIST, and CIFAR10.

Table 3.8 Selected values of the adversarial perturbation ε for the NAS, for MNIST, Fashion-MNIST, and CIFAR10. The table also reports the values for ε_{low} and ε_{high} for the *Two EPS* search. The *One EPS* column denotes a search that uses only one value of ε, whereas the *Two EPS* column denotes a search that uses both low and high values of ε. Take into account that a simple dataset like the MNIST needs a significant adversarial perturbation to have an effect on the DNN robustness. On the other hand, a smaller perturbation is already enough to incorrectly categorize a particular set of inputs on a more complicated dataset like the CIFAR10.

	Two EPS ε_{low}	**One EPS ε**	**Two EPS ε_{high}**
MNIST	3e-3	3e-2	1e-1
FMNIST	1e-3	1e-2	3e-2
CIFAR10	3e-5	3e-4	1e-3

Results on partially-trained DNNs

As discussed in Section 3.4.5, during the NAS iterations the models are only partially trained to reduce the overall exploration time. The approach and setting are the same as in Section 3.4.5, and therefore not furtherly discussed.

Fig. 3.29 reports the results of the NAS with fast robustness evaluation. The Pareto-optimal solutions are discovered in the latest generations of the algorithm, whereas the earliest generate sub-optimal DNN solutions. It should be noted that the

latest DNN generations for the CIFAR10 dataset are less robust to the PGD attack but still fall within the Pareto-frontier because of their low energy consumption (see pointer ①). The Pareto-frontier selection automatically eliminates some candidate DNNs discovered in the initial generations since they are very vulnerable to the PGD attack, as pointed out by pointer ②.

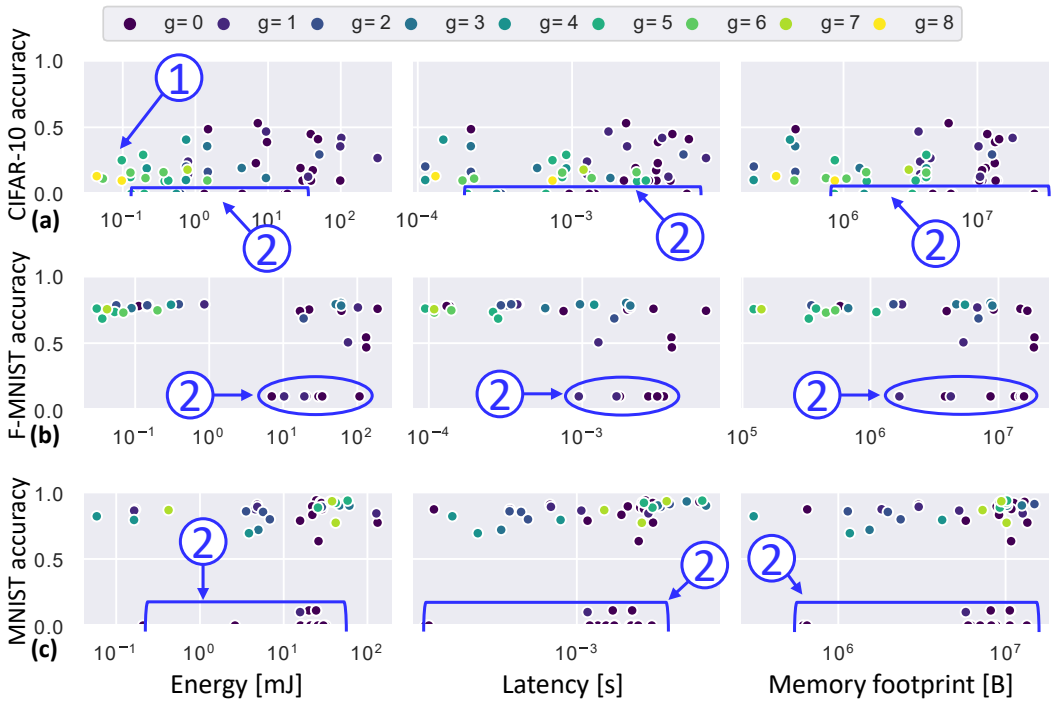


Fig. 3.29 DNN robustness of partially-trained DNNs under PGD attack, showing tradeoffs w.r.t. energy, latency, and memory footprint. (a) Results for CIFAR10. (b) Results for Fashion-MNIST. (c) Results for MNIST.

Results on fully-trained DNNs

The Pareto-optimal DNNs that were selected in the previous step have been fully trained to get an accurate robustness evaluation. The DNNs targeting the MNIST and Fashion-MNIST datasets have been trained for 100 epochs, while the DNNs for the CIFAR10 dataset have been trained for 300 epochs. Tradeoffs between the design objectives are evident in the findings shown in Fig. 3.30.

As shown by pointer ①, the framework's Pareto-optimal solution for the CIFAR10 dataset achieves 86.07% accuracy while using 38.63 mJ of energy, 11.85 MB of memory, and 4.47 ms of latency. The Fashion-MNIST dataset solution pointed

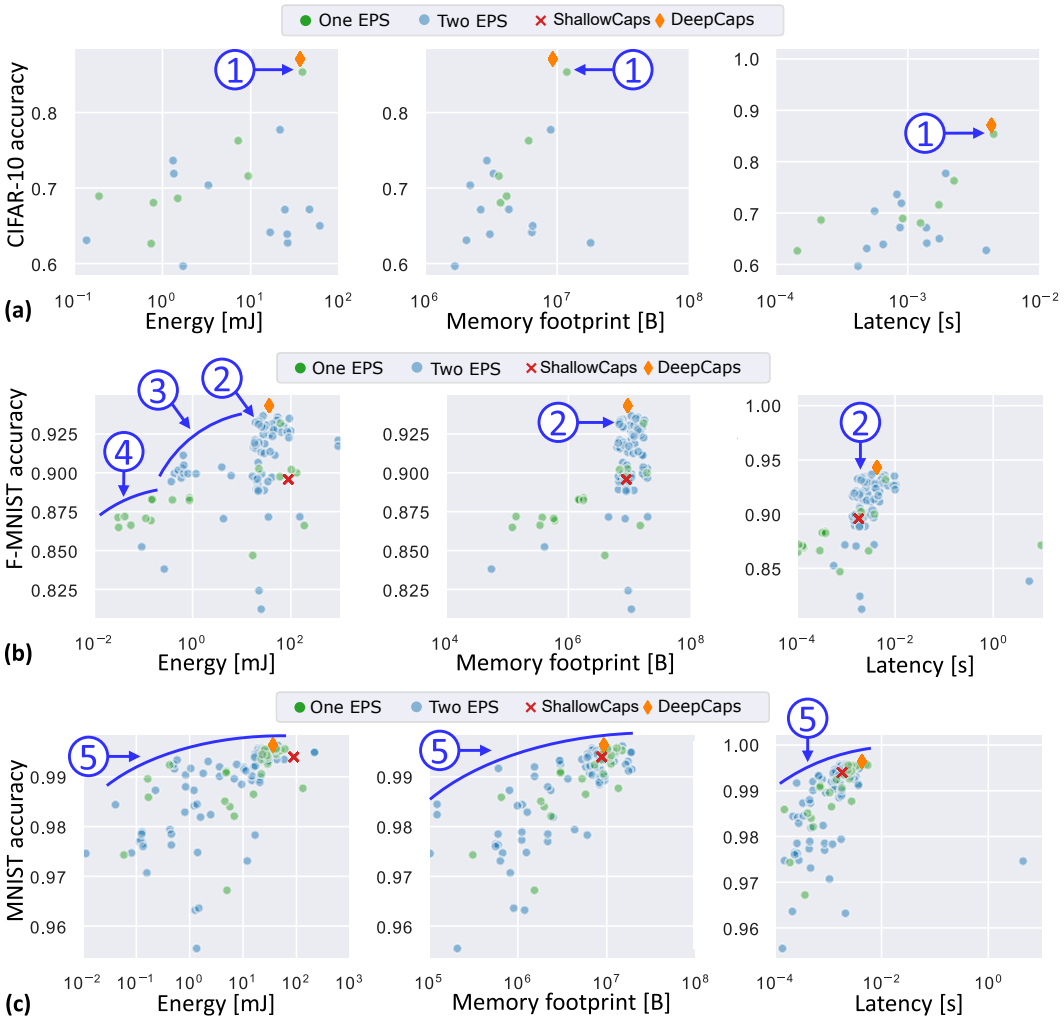


Fig. 3.30 Robustness of fully-trained Pareto-optimal DNNs, showing tradeoffs w.r.t. hardware efficiency. (a) Results for CIFAR10. (b) Results for Fashion-MNIST. (c) Results for MNIST.

out in ② achieves an accuracy of 93.40% while requiring 6.40 ms of latency, 61.19 mJ of energy, and 16.82 MB of memory. It should be noted that while the *One EPS* search discovers more interesting low-energy solutions (see pointer ④), the *Two EPS* search finds Pareto-optimal solutions in the middle range of energy (see pointer ③). By leveraging trade-offs between many objectives, the Pareto-optimal DNNs' search for MNIST spans a more diversified range of values (see pointer ⑤).

In Fig. 3.31, we compare the discovered solutions (One EPS setting) with the ShallowCaps model, the DeepCaps model, and the models found with the origi-

nal NASCaps framework of Section 3.4, i.e., without adversarial robustness as an objective.

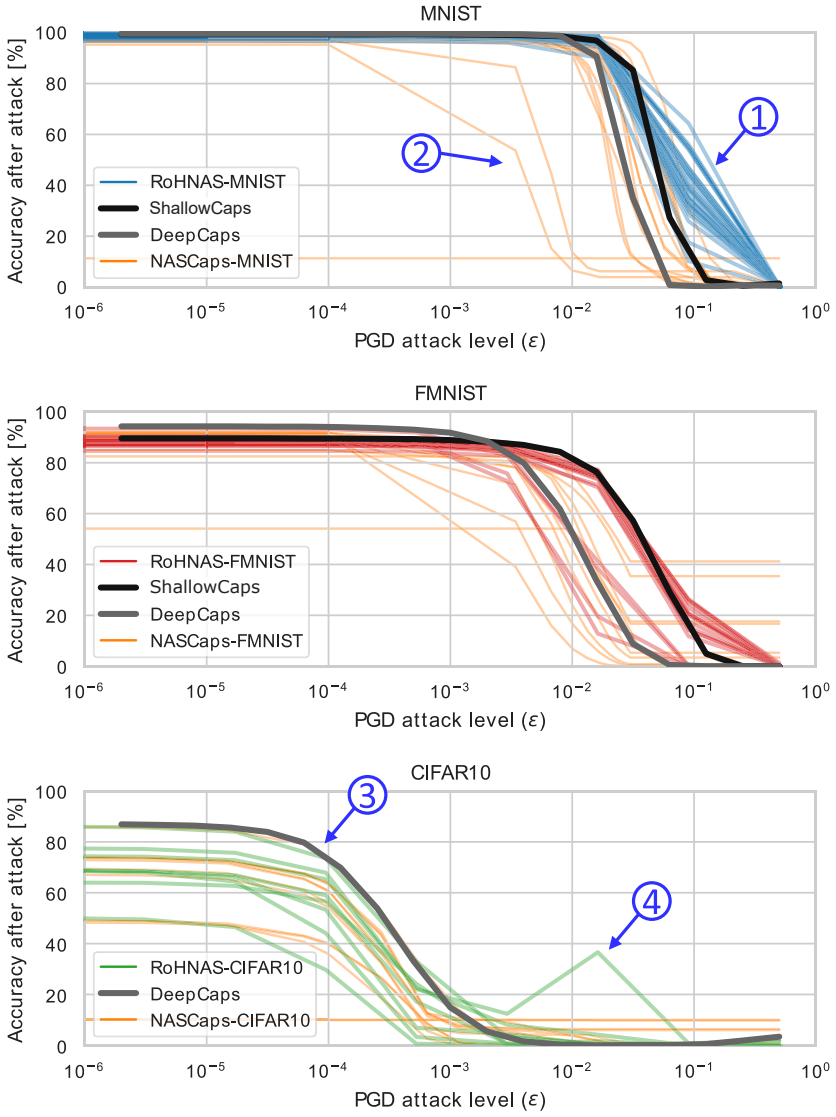


Fig. 3.31 Evaluation of the modified framework, compared to other state-of-the-art and NASCaps-discovered architectures.

The Pareto-optimal solutions produced for the MNIST dataset using the re-designed framework are very resilient across a wide range of perturbation ε (see pointer ①). In fact, the accuracy begins to decline at an ε that is almost one order of magnitude greater than for the original NASCaps (see pointer ②). The robustness of the Pareto-optimal DNNs chosen with the framework for the Fashion-MNIST is very similar to that of the ShallowCaps architecture. For the CIFAR10 dataset, the

discovered Pareto-optimal DNNs behave similarly to the DeepCaps for low values of ε (see pointer ③). However, one particular Pareto-optimal solution provides notable robustness also with stronger adversarial perturbation (see pointer ④).

Fig. 3.32 shows the comparison of the modified framework with the original NASCaps framework, the ShallowCaps, and the DeepCaps models when the *Two EPS* setting is used. Especially for the MNIST and Fashion-MNIST datasets (see pointer ① in Fig. 3.32), w.r.t. the *One EPS* setting, the framework returns wider levels of robustness.

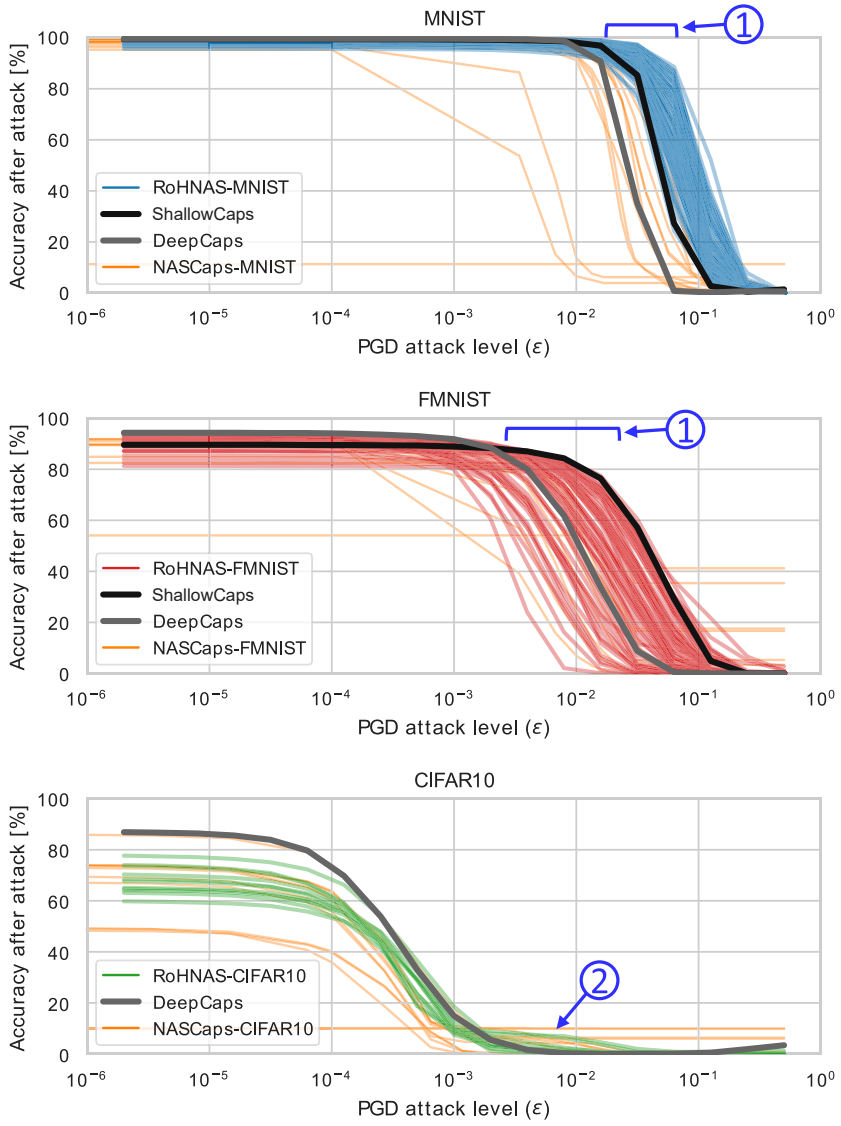


Fig. 3.32 Evaluation of the modified framework with the *Two EPS* setting, compared to other state-of-the-art and NASCaps-discovered architectures.

3.5.3 Conclusions

The strategy proposed in this section allows jointly optimizing traditional CNNs and CapsNets for hardware efficiency (latency, energy, and memory footprint) and robustness against adversarial attacks. Our optimizations for reducing the search space and the exploration time allow finding a set of networks that are Pareto-optimal w.r.t. the above-discussed objectives, in a fast fashion. In our experiments, 900 different DNN models have been evaluated, using 2,000 GPU hours with our fast training settings. Thanks to this NASCaps version, the deployment of robust DNNs in resource-constrained IoT/neuromorphic edge devices is made possible.

Chapter 4

Conclusions and Future Works

This thesis shows how the three aspects of workload, peak performance, and efficiency are crucial for HW acceleration of neural networks, taking Winograd's algorithm for convolutions and capsule networks as practical cases.

Chapter 2 presented a novel quantization technique to deploy quantized Winograd convolutions on $4\times$ 4 tiles, and an investigation on how to integrate the algorithm in an industrial HW accelerator for AI applications. The first part of the chapter presented the tap-wise quantization algorithm to enable efficient quantized Winograd on $4\times$ 4 tiles F_4. Using 8-bits integers for the feature maps and weights and 10-bits integers in the Winograd domain, the F_4 Winograd network achieves the same accuracy as the FP32 baseline for ResNet-20 and VGG-nagadomi on the CIFAR-10 benchmark and for ResNet-50 on the ImageNet classification task. The proposed method outperforms the state-of-the-art integer-only and F_4-aware quantization methods on all the tested networks and tasks. Furthermore, the second part of the chapter presents a custom HW extension to process F_4 integer Winograd layers efficiently and its integration into an industrial-grade AI accelerator. Our proposed system outperforms NVDLA with its Winograd F_2 extension by 1.5 to $3.3\times$ at the same compute throughput and bandwidth constraints. This is achieved with the higher computational reduction from F_4, optimized bandwidth requirements by on-the-fly transformations, and higher utilization thanks to the optimized dataflow. The proposed hardware extensions have a small area (6.1% of the core area) and power (17% compared to the MatMul engine) overhead over the baseline architecture while achieving up to $3.42\times$ speed-up on compute-intensive convolutional layers.

An extensive evaluation over several state-of-the-art computer-vision benchmarks revealed up to 1.83× end-to-end inference speed-up and 1.85× energy efficiency improvement.
One potential direction for future research is to investigate the combination of quantization with orthogonal techniques, such as pruning, to achieve an even greater reduction of the overall computational workload. It is worth noting that throughout the exploration of these techniques and methodologies, it is essential to always ensure that the theoretical gains in, e.g., computational reduction, can effectively translate into speed-up or energy savings when executed on an HW accelerator. A significant critical issue faced was the long time it took to develop custom HW units for the algorithm under study and to integrate them into the accelerator, two steps necessary to evaluate performances on different layers and networks then. Part of this problem was solved by developing a simulator, which allows design-space-exploration and makes architectural choices much faster and possible before starting the whole HW design flow. However, this does not detract from the time-consuming development and integration of dedicated units. Moreover, integrating additional resources for each algorithm risks leading to an unsustainable area explosion. Therefore, from a future perspective, we believe it is crucial to focus on developing new architectures for DSAs that are more flexible, with easily programmable or reconfigurable units based on the requirements of each algorithm or workload.

Chapter 3 focused on our efforts to push forward the research field on capsule networks, making them more HW-efficient and finding new capsule-based models. The first part of the chapter describes a specialized framework for quantizing CapsNets, called Q-CapsNets. We exploited the peculiar features of CapsNets, occurring during the dynamic routing, for designing a quantization methodology that enables further precision reduction of the wordlength while a certain accuracy loss is tolerated. Our Q-CapsNets framework produces compact yet accurate quantized CapsNet models. Hence, it represents the first step towards designing energy-efficient CapsNets, and could potentially open new avenues towards the large-scale adoption of CapsNets for inference in a resource-constrained scenario. As described in detail in Chapter 3, the purpose of the proposed quantization framework is to enable quick exploration of CapsNets quantization-accuracy tradeoffs to drive the design of HW architectures. With this in mind, it was decided to use a simple post-training quantization approach with a search in a pruned exploration space. As an extension of the framework, the more complex quantization-aware training could be applied. At the cost of more

computational resources and training time, it could guarantee a higher wordlength reduction for the same accuracy. Note that, at the time of the project, machine learning frameworks such as PyTorch and Tensorflow had no support for quantized inference or training.

The second part of Chapter 3 presents NASCaps, a framework for the NAS of convolutional CapsNets. The optimization goals of our framework are network accuracy and hardware efficiency, expressed in terms of energy consumption, memory footprint, and latency, when executed on the specialized hardware accelerators. We performed a large-scale NAS using GPU-HPC nodes with multiple Tesla V100 GPUs and found interesting DNN solutions that are hardware-efficient yet highly accurate when compared to SOTA solutions. Our framework is even more beneficial when the design times are short, training resources at the design center are limited, and the DNN design is subjected to short training durations. Our NASCaps framework can ease the deployment of DNNs based on capsule layers in resource-constrained IoT/edge devices.

It is worth noting that despite the initial surge in popularity, capsule networks have not been able to establish themselves as direct substitutes or competitors to the more traditional CNNs. Therefore, a significant contribution to this research area would involve primarily exploring applications where the unique structure of capsule networks can offer a competitive advantage. By identifying specific domains or problem spaces where the inherent properties of CapsNets, such as hierarchical representation and dynamic routing, can be leveraged effectively, researchers can uncover the true potential and advantages of this alternative architecture. This exploration could involve investigating scenarios where capturing spatial relationships, pose estimation, or viewpoint invariance play pivotal roles. Demonstrating the superiority of capsule networks in such application areas would enhance their standing and facilitate their adoption as viable alternatives to CNNs.

References

[1] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. AI and ML accelerator survey and trends. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*, pages 1–10. IEEE, 2022.

[2] Jathushan Rajasegaran et al. Deepcaps: Going deeper with capsule networks. In *CVPR*, 2019.

[3] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[4] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[5] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *NIPS*, 2017.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *Proc. IEEE CVPR*, pages 770–778, 2015.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[8] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. *Advances in neural information processing systems*, 13, 2000.

[9] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[11] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[12] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification With Deep Convolutional Neural Networks. In *Adv. NIPS*, 2012.

[14] Kaiyuan Guo, Song Han, Song Yao, Yu Wang, Yuan Xie, and Huazhong Yang. Software-hardware codesign for efficient neural network acceleration. *IEEE Micro*, 37(2):18–25, 2017.

[15] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[16] 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.

[17] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.

[18] Pramod Udupa, Gopinath Mahale, Kiran Kolar Chandrasekharan, and Sehwan Lee. Accelerating depthwise convolution and pooling operations on z-first storage cnn architectures. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[19] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[20] Frank Rosenblatt. *The Perceptron: A Perceiving and Recognizing Automaton (Project PARA). Report No. 85-460-1*. Cornell Aeronautical Laboratory, 1957.

[21] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[23] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211–252, 2015.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[26] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[27] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[28] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[29] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 4278–4284. AAAI Press, 2017.

[30] 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.

[31] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R. Manmatha, Mu Li, and Alexander J. Smola. Resnest: Split-attention networks. *CoRR*, abs/2004.08955, 2020.

[32] Tal Ridnik, Hussam Lawen, Asaf Noy, and Itamar Friedman. Tresnet: High performance gpu-dedicated architecture. *CoRR*, abs/2003.13630, 2020.

[33] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

[34] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.

[35] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part I*, volume 12346 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2020.

[36] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[37] Mark Chen, Alec Radford, Rewon Child, Jeffrey Wu, Heewoo Jun, David Luan, and Ilya Sutskever. Generative pretraining from pixels. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 1691–1703. PMLR, 2020.

[38] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE*, 105(12):2295–2329, 2017.

[39] R James. Intel avx-512 instructions. *Intel*, 06 2017.

[40] Adam Paszke et al. Automatic differentiation in pytorch. 2017.

[41] Martin Abadi et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In Kien A. Hua, Yong Rui, Ralf Steinmetz, Alan Hanjalic, Apostol Natsev, and Wenwu Zhu, editors, *Proceedings of the ACM International Conference on Multimedia, MM '14, Orlando, FL, USA, November 03 - 07, 2014*, pages 675–678. ACM, 2014.

[43] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[44] *NVIDIA TESLA V100 GPU ARCHITECTURE*, 2017.

[45] *NVIDIA A100 Tensor Core GPU Architecture*, 2020.

[46] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), Oct 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[47] A. Vasudevan, A. Anderson, and D. Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 19–24, 2017.

[48] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, aug 1969.

[49] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In Stefan Wermter, Cornelius Weber, Wlodzislaw Duch, Timo Honkela, Petia D. Koprinkova-Hristova, Sven Magg, Günther Palm, and Alessandro E. P. Villa, editors, *Artificial Neural Networks and Machine Learning - ICANN 2014 - 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings*, volume 8681 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 2014.

[50] Michaël Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[51] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[52] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *Proc. IEEE CVPR*, pages 4013–4021, 2016.

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

[54] 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. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 49:269–284, 02 2014.

[55] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[56] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37:12–21, 2017.

[57] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. A 240 g-ops/s mobile coprocessor for deep neural networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 696–701, June 2014.

[58] V. Sriram, D. Cox, K. H. Tsoi, and W. Luk. Towards an embedded biologically-inspired machine vision processor. In *2010 International Conference on Field-Programmable Technology*, pages 273–278, Dec 2010.

[59] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. A massively parallel coprocessor for convolutional neural networks. In *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 53–60, July 2009.

[60] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, jun 2017.

[61] Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic CNN accelerator simulator. *CoRR*, abs/1811.02883, 2019.

[62] C. Luo, Y. Wang, W. Cao, P. H. W. Leong, and L. Wang. Rna: An accurate residual network accelerator for quantized and reconstructed deep neural networks. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 60–603, 2018.

[63] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, June 2015.

[64] L. Cavigelli and L. Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, Nov 2017.

[65] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal. Memory-centric accelerator design for convolutional neural networks. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 13–19, Oct 2013.

[66] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017.

[67] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[68] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, Dec 2014.

[69] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *Hot Chips Symposium*, pages 1–44, 2019.

[70] Karam Chatha. Qualcomm® cloud al 100 : 12tops/w scalable, high performance and low latency deep learning inference accelerator. In *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*, pages 1–19. IEEE, 2021.

[71] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.

[72] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025, 2016.

[73] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115, 2014.

[74] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Accurate post training quantization with small calibration sets. In *International Conference on Machine Learning*, pages 4466–4475. PMLR, 2021.

[75] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems*, 32, 2019.

[76] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018. IEEE, 2019.

[77] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *International conference on machine learning*, pages 7543–7552. PMLR, 2019.

[78] 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.

[79] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(11):5784–5789, 2018.

[80] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[81] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[82] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[83] Charbel Sakr and Naresh Shanbhag. Per-tensor fixed-point quantization of the back-propagation algorithm. In *ICLR*, 2019.

[84] Maithra Raghu et al. On the expressive power of deep neural networks. In *ICML*, 2017.

[85] Darryl Dexu Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2849–2858. JMLR.org, 2016.

[86] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Haq: Hardware-aware automated quantization with mixed precision. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8604–8612, 2019.

[87] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

[88] Y. Umuroglu, L. Rasnayake, and M. Själander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. In *2018 28th*

International Conference on Field Programmable Logic and Applications (FPL), pages 307–3077, 2018.

[89] P. Judd, J. Albericio, and A. Moshovos. Stripes: Bit-serial deep neural network computing. *IEEE Computer Architecture Letters*, 16(1):80–83, 2017.

[90] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2019.

[91] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos. Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[92] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[93] S. Ryu, H. Kim, W. Yi, and J. Kim. Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[94] Enrico Reggiani, Alessandro Pappalardo, Max Doblas, Miquel Moreto, Mauro Olivieri, Osman Sabri Unsal, and Adrián Cristal. Mix-gemm: An efficient hw-sw architecture for mixed-precision quantized deep neural networks inference on edge devices. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1085–1098, 2023.

[95] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[96] Song Han, Jeff Pool, m john Tran, William J Dally, John Tran, William J Dally, m john Tran, and William J Dally. Learning Both Weights and Connections for Efficient Neural Networks. *NIPS*, 2015.

[97] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In Xianghua Xie, Mark W. Jones, and Gary K. L. Tam, editors, *Proceedings of the British Machine Vision Conference 2015, BMVC 2015, Swansea, UK, September 7-10, 2015*, pages 31.1–31.12. BMVA Press, 2015.

[98] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1398–1406, 2017.

[99] F. Tung and G. Mori. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[100] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 815–832, Cham, 2018. Springer International Publishing.

[101] H. Cai, J. Lin, Y. Lin, Z. Liu, K. Wang, T. Wang, L. Zhu, and S. Han. Automl for architecting efficient and specialized neural networks. *IEEE Micro*, 40(1):75–82, 2020.

[102] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han. Apq: Joint search for network architecture, pruning and quantization policy. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2075–2084, 2020.

[103] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, 2017.

[104] Haichuan Yang, Yuhao Zhu, and Ji Liu. ECC: platform-independent energy-constrained deep neural network compression via a bilinear regression model. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 11206–11215. Computer Vision Foundation / IEEE, 2019.

[105] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 535–541, New York, NY, USA, 2006. Association for Computing Machinery.

[106] L.J. Ba and R. Caruana. Do deep nets really need to be deep? *Advances in Neural Information Processing Systems*, 3:2654–2662, 01 2014.

[107] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv preprint arXiv:1503.02531*, 2015.

[108] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[109] J. Yim, D. Joo, J. Bae, and J. Kim. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *2017 IEEE*

Conference on Computer Vision and Pattern Recognition (CVPR), pages 7130–7138, 2017.

[110] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu. Deep mutual learning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4320–4328, 2018.

[111] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807, 2017.

[112] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. Speeding-up convolutional neural networks using finetuned cp-decomposition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[113] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[114] Hieu Pham et al. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

[115] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[116] Dimitrios Stamoulis et al. Single-path NAS: designing hardware-efficient convnets in less than 4 hours. In *ECML/PKDD*, 2019.

[117] Mitchell P. Marcus et al. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 1993.

[118] P. Achararit et al. Apnas: Accuracy-and-performance-aware neural architecture search for neural hardware accelerators. *IEEE Access*, 2020.

[119] Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. Attentivenas: Improving neural architecture search via attentive sampling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6418–6427, 2021.

[120] Qing Lu et al. On neural architecture search for resource-constrained hardware platforms. *CoRR*, abs/1911.00105, 2019.

[121] Weiwen Jiang et al. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *DAC*, 2019.

[122] Weiwen Jiang et al. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE TCAD*, 2020.

[123] Zichao Guo et al. Single path one-shot neural architecture search with uniform sampling. In *ECCV*, 2020.

[124] Li Lyna Zhang et al. Hardware-aware one-shot neural architecture search in coordinate ascent framework. *CoRR*, abs/1910.11609, 2019.

[125] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[126] Renzo Andri, Beatrice Bussolino, Antonio Cipolletta, Lukas Cavigelli, and Zhe Wang. Going further with winograd convolutions: Tap-wise quantization for efficient inference on 4x4 tiles. *IEEE Micro*, 2022.

[127] Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1969–1979, 2020.

[128] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[129] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[130] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.

[131] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. *arXiv:1804.02767*, 2018.

[132] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[133] Barbara Barabasz, Andrew Anderson, Kirk M Soodhalter, and David Gregg. Error analysis and improving the accuracy of Winograd convolution for deep neural networks. *ACM Transactions on Mathematical Software (TOMS)*, 46(4):1–33, 2020.

[134] Javier Fernandez-Marques, Paul N Whatmough, Andrew Mundy, and Matthew Mattina. Searching for winograd-aware quantized networks. *MLSys*, 2021.

[135] Barbara Barabasz and David Gregg. Winograd Convolution for DNNs: Beyond Linear Polynomials. In Mario Alviano, Gianluigi Greco, and Francesco Scarcello, editors, *AI*IA 2019 – Advances in Artificial Intelligence*, pages 307–320, Cham, 2019. Springer International Publishing.

[136] Juan Yepez and Seok-Bum Ko. Stride 2 1-D, 2-D, and 3-D winograd for convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):853–863, 2020.

[137] Chen Yang, Yizhou Wang, Xiaoli Wang, and Li Geng. A stride-based convolution decomposition method to stretch CNN acceleration algorithms for efficient and flexible hardware implementation. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(9):3007–3020, 2020.

[138] Yulin Zhao, Donghui Wang, Leiou Wang, and Peng Liu. A faster algorithm for reducing the computational complexity of convolutional neural networks. *Algorithms*, 11(10):159, 2018.

[139] Syed Asad Alam, Andrew Anderson, Barbara Barabasz, and David Gregg. Winograd Convolution for Deep Neural Networks: Efficient Point Selection. *arXiv preprint arXiv:2201.10369*, 2022.

[140] Wenshuo Li, Hanting Chen, Mingqiang Huang, Xinghao Chen, Chunjing Xu, and Yunhe Wang. Winograd Algorithm for AdderNet. In *International Conference on Machine Learning*, pages 6307–6315. PMLR, 2021.

[141] Xingyu Liu, Jeff Pool, Song Han, and William J Dally. Efficient sparse-winograd convolutional neural networks. *arXiv preprint arXiv:1802.06367*, 2018.

[142] Sheng Li, Jongsoo Park, and Ping Tak Peter Tang. Enabling sparse winograd convolution by native pruning. *arXiv preprint arXiv:1702.08597*, 2017.

[143] Jiong Gong, Haihao SHEN, Xiao Dong Lin, and Xiaoli Liu. Method and apparatus for keeping statistical inference accuracy with 8-bit winograd convolution, 2018.

[144] Guangli Li, Lei Liu, Xueying Wang, Xiu Ma, and Xiaobing Feng. Lance: efficient low-precision quantized winograd convolution for neural networks based on graphics processing units. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3842–3846. IEEE, 2020.

[145] Lingchuan Meng and John Brothers. Efficient winograd convolution via integer arithmetic. *arXiv preprint arXiv:1901.01965*, 2019.

[146] Zhi-Gang Liu and Matthew Mattina. Efficient Residue Number System Based Winograd Convolution. In *European Conference on Computer Vision*, pages 53–68. Springer, 2020.

[147] Guangli Li, Zhen Jia, Xiaobing Feng, and Yida Wang. LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs. In *50th International Conference on Parallel Processing*, pages 1–11, 2021.

[148] Xinheng Liu, Yao Chen, Cong Hao, Ashutosh Dhar, and Deming Chen. WinoCNN: Kernel sharing Winograd systolic array for efficient convolutional neural network acceleration on FPGAs. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 258–265. IEEE, 2021.

[149] Liqiang Lu and Yun Liang. SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[150] Tao Yang, Zhezhi He, Tengchuan Kou, Qingzheng Li, Qi Han, Haibao Yu, Fangxin Liu, Yun Liang, and Li Jiang. BISWSRBS: A Winograd-based CNN Accelerator with a Fine-grained Regular Sparsity Pattern and Mixed Precision Quantization. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 14(4):1–28, 2021.

[151] Shihang Wang, Jianghan Zhu, Qi Wang, Can He, and Terry Tao Ye. Customized Instruction on RISC-V for Winograd-Based Convolution Acceleration. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 65–68. IEEE, 2021.

[152] Athanasios Xygkis, Dimitrios Soudris, Lazaros Papadopoulos, Sofiane Yous, and David Moloney. Efficient winograd-based convolution kernel implementation on edge devices. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[153] Gopinath Mahale, Pramod Udupa, Kiran Kolar Chandrasekharan, and Sehwan Lee. WinDConv: A Fused Datapath CNN Accelerator for Power-Efficient Edge Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4278–4289, 2020.

[154] NVIDIA. Nvdla primer - nvdla documentation.

[155] Prateeth Nayak, David Zhang, and Sek Chai. Bit efficient quantization for deep neural networks. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 52–56. IEEE, 2019.

[156] Junhong Liu, Dongxu Yang, and Junjie Lai. Optimizing Winograd-Based Convolution with Tensor Cores. In *50th International Conference on Parallel Processing*, pages 1–10, 2021.

[157] Roberto L Castro, Diego Andrade, and Basilio B Fraguela. OpenCNN: A Winograd Minimal Filtering Algorithm Implementation in CUDA. *Mathematics*, 9(17):2033, 2021.

[158] Sumin Kim, Gunju Park, and Youngmin Yi. Performance Evaluation of INT8 Quantized Inference on Mobile GPUs. *IEEE Access*, 9:164245–164255, 2021.

[159] Dongsheng Li, Dan Huang, Zhiguang Chen, and Yutong Lu. Optimizing Massively Parallel Winograd Convolution on ARM Processor. In *50th International Conference on Parallel Processing*, pages 1–12, 2021.

[160] Partha Maji, Andrew Mundy, Ganesh Dasika, Jesse Beu, Matthew Mattina, and Robert Mullins. Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus. In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 1–5. IEEE, 2019.

[161] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv:1308.3432*, 2013.

[162] Sambhav R Jain, Albert Gural, Michael Wu, and Chris H Dick. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. *arXiv preprint arXiv:1903.08066*, 2019.

[163] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[164] Norm Jouppi. Google supercharges machine learning tasks with TPU custom chip, 2016.

[165] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[166] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, Renton, WA, July 2019. USENIX Association.

[167] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[168] James E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual Symposium on Computer Architecture*, ISCA '82, page 112–119, Washington, DC, USA, 1982. IEEE Computer Society Press.

[169] Zhengrong Wang and Tony Nowatzki. Stream-based memory access specialization for general purpose processors. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 736–749, 2019.

[170] Nagadomi. *kaggle-cifar10-torch7*. 2014.

[171] Luca Saglietti and Lenka Zdeborová. Solvable model for inheriting the regularization through knowledge distillation. In *Mathematical and Scientific Machine Learning*, pages 809–846. PMLR, 2022.

[172] Barbara Barabasz. Quantized Winograd/Toom-Cook Convolution for DNNs: Beyond Canonical Polynomials Base. *arXiv preprint arXiv:2004.11077*, 2020.

[173] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018.

[174] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, 461:370–403, 2021.

[175] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladrish Chatterjee, Nan Jiang, and David Nellans. Need for speed: Experiences building a trustworthy system-level gpu simulator. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 868–880, 2021.

[176] Lukas Steiner, Matthias Jung, and Norbert Wehn. Exploration of ddr5 with the open-source simulator dramsys. In *MBMV 2021; 24th Workshop*, pages 1–11. VDE, 2021.

[177] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE, 2015.

[178] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[179] Alberto Marchisio, Beatrice Bussolino, Alessio Colucci, Maurizio Martina, Guido Masera, and Muhammad Shafique. Q-capsnets: A specialized framework for quantizing capsule networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[180] Alberto Marchisio, Andrea Massa, Vojtech Mrazek, Beatrice Bussolino, Maurizio Martina, and Muhammad Shafique. Nascaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[181] Alberto Marchisio, Vojtech Mrazek, Andrea Massa, Beatrice Bussolino, Maurizio Martina, and Muhammad Shafique. Rohnas: A neural architecture search framework with conjoint optimization for adversarial robustness and hardware efficiency of convolutional and capsule networks. *IEEE Access*, 10:109043–109055, 2022.

[182] Alberto Marchisio, Muhammad Abdullah Hanif, and Muhammad Shafique. Capsacc: An efficient hardware accelerator for capsulenets with data reuse. In *DATE*, 2019.

[183] Patrick Kwabena Mensah, Adebayo Felix Adekoya, Mighty Abra Ayidzoe, and Edward Y. Baagyire. Capsule networks - A survey. *J. King Saud Univ. Comput. Inf. Sci.*, 34(1):1295–1310, 2022.

[184] Jiawei Li, Qichen Zhao, Nan Li, Lin Ma, Xuan Xia, Xiaoguang Zhang, Ning Ding, and Nannan Li. A survey on capsule networks: Evolution, application, and future development. In *2021 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*, pages 177–185, 2021.

[185] Geoffrey E. Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[186] Dilin Wang and Qiang Liu. An optimization view on dynamic routing between capsules. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018.

[187] Sameera Ramasinghe, C. D. Athuraliya, and Salman H. Khan. A context-aware capsule network for multi-label classification. In Laura Leal-Taixé and Stefan Roth, editors, *Computer Vision - ECCV 2018 Workshops - Munich, Germany, September 8-14, 2018, Proceedings, Part III*, volume 11131 of *Lecture Notes in Computer Science*, pages 546–554. Springer, 2018.

[188] Liheng Zhang, Marzieh Edraki, and Guo-Jun Qi. Cappronet: Deep feature learning via orthogonal projections onto capsule subspaces. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 5819–5828, 2018.

[189] Miguel Costa, Diogo Costa, Tiago Gomes, and Sandro Pinto. Shifting capsule networks from the cloud to the deep edge. *CoRR*, abs/2110.02911, 2021.

[190] M. Shafique et al. Robust machine learning systems: Challenges,current trends, perspectives, and the road ahead. *IEEE Design & Test*, 2020.

[191] Christian Szegedy et al. Intriguing properties of neural networks. In *ICLR*, 2014.

[192] Xiaoyong Yuan et al. Adversarial examples: Attacks and defenses for deep learning. *IEEE Trans. on Neural Networks and Learning Systems*, 2019.

[193] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[194] Aleksander Madry et al. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[195] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *ICLR (Workshop)*, 2017.

[196] Amira Guesmi, Ihsen Alouani, Khaled N. Khasawneh, Mouna Baklouti, Tarek Frikha, Mohamed Abid, and Nael B. Abu-Ghazaleh. Defensive approximation: securing cnns using approximate computing. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 990–1003. ACM, 2021.

[197] Qi Liu, Tao Liu, Zihao Liu, Yanzhi Wang, Yier Jin, and Wujie Wen. Security analysis and enhancement of model compressed deep learning systems under adversarial attacks. In Youngsoo Shin, editor, *23rd Asia and South Pacific Design Automation Conference, ASP-DAC 2018, Jeju, Korea (South), January 22-25, 2018*, pages 721–726. IEEE, 2018.

[198] Minghao Guo, Yuzhe Yang, Rui Xu, and Z. Liu. When nas meets robustness: In search of robust architectures against adversarial attacks. In *CVPR*, 2020.

[199] Danilo Vasconcellos Vargas and Shashank Kotyan. Evolving robust neural architectures to defend from adversarial attacks. *CoRR*, abs/1906.11667, 2019.

[200] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *ArXiv*, 2017.

[201] Shachar Gluska and Mark Grobman. Exploring neural networks quantization via layer-wise quantization analysis. *arXiv preprint arXiv:2012.08420*, 2020.

[202] Chen Tang, Kai Ouyang, Zhi Wang, Yifei Zhu, Yaowei Wang, Wen Ji, and Wenwu Zhu. Mixed-precision neural network quantization via learned layer-wise importance. *arXiv preprint arXiv:2203.08368*, 2022.

[203] Yanan Sun, Bing Xue, Mengjie Zhang, and Gary G. Yen. Automatically designing cnn architectures using genetic algorithm for image classification. *IEEE transactions on cybernetics*, 2020.

[204] Weiwen Jiang et al. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *DAC*, 2019.

[205] W. Jiang et al. Device-circuit-architecture co-exploration for computing-in-memory neural accelerators. *IEEE Transactions on Computers*, 2020.

[206] Zhichao Lu, Kalyanmoy Deb, and Vishnu Naresh Boddeti. Muxconv: Information multiplexing in convolutional neural networks. *ArXiv*, 2020.

[207] Dimitrios Stamoulis et al. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In *ECML/PKDD*, 2019.

[208] Aaron Harlap et al. Pipedream: Fast and efficient pipeline parallel dnn training. *ArXiv*, 2018.

[209] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 1895.

[210] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 2002.

[211] Yuval Netzer et al. Reading digits in natural images with unsupervised feature learning. *NIPS*, 2011.

[212] Chih-Hong Cheng, Frederik Diehl, Gereon Hinz, Yassine Hamza, Georg Nührenberg, Markus Rickert, Harald Ruess, and Michael Truong-Le. Neural networks for safety-critical applications—challenges, experiments and perspectives. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1005–1006. IEEE, 2018.