POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONICS AND
TELECOMMUNICATIONS

Master Degree Course in Electronic Engineering

Master Degree Thesis

FPGA-based Deep Learning Inference Acceleration at the Edge

Supervisor
Prof. Mihai Teodor Lazarescu

Candidate
Andrea Casale

Anno accademico 2020 – 2021

Acknowledgements

Before proceeding with the discussion, I would like to dedicate a few lines to all those who have been close to me over the years.

First of all, I would like to express a special thanks to my supervisor, prof. Mihai Lazarescu, who helped me, with his infinite availability and kindness, throughout the writing of my thesis. I would like to thank him for his valuable advice and for promptly suggesting me the right changes to make to my work.

I would also like to thank all the people who have encouraged me along the way. Thank you to all my friends, especially Alessandro, Marco, Luigi, Matteo, Andrea, Domenico, Martina, Giacomo, Giovanni and Giulia, who supported and put up with me in the most difficult moments, encouraging to not give up. Thanks for always being there and for your patience, you are like a second family for me. A special thanks to Laura, for the remote support provided to me, without ever asking anything in return. In the same way, I would like to thank all the people I have met during my time at university, with whom I have shared moments of study and fun. To mention you all would be impossible, but each of you has been equally important. A special thanks to Sandro, Gianluca, Pellegrino, Simone, Luigi, Davide, Marco and Roberta, with whom I shared the entire course of my master's studies.

A final thanks goes out to my parents, without whom I could never have achieved all this. I will never stop thanking you for giving me the opportunity to achieve this result. Thanks to my cousins Davide and Eleonora, my uncles and my grandmother for being there for me, especially in the most difficult moments.

A Pako.

Abstract

Over the last few years, with the exponential increase in the amount of data available, *Deep Neural Networks* (DNNs) have become the most widely used computational model in solving complex problems for most of applications, due to the extremely high level of accuracy they are capable of attaining compared to that of traditional *Machine Learning* (ML) algorithms, even better than human capabilities.

However, in order to achieve this level of accuracy on the results produced, the number of layers and neurons of DNNs have become increasingly high over time, resulting in a huge demand of computational and storage resources to handle the execution on the increasing number of operations and parameters required in DL processing in real-time, both during the *training* and *inference* phases.

While currently the training requires too high computational performance and arithmetic precision to think of an execution different than on the cloud, the current trend is to move the inference execution from the cloud to the edge, close to where the data are collected, saving the latency and energy consumption required to transfer the processing to the server. However, while the execution at the cloud is based on maximizing performance, the requirements for inference processing at the edge can be very different, focusing on high throughput, low latency and low energy consumption. Thus, maintaining a high accuracy for DNN models implemented on devices with a limited amount of resources results to be a challenging task.

To address this problem, a large number of specialized *Artificial Neural Network* (ANN) *accelerators* based on different processing units have been proposed over time, designed to accelerate the inference processing in hardware by exploiting the high level of inherent parallelism exhibited by DNN models. In general, the standard internal architecture of each of these processing units is composed of an array of *Processing Elements* (PEs) responsible for the execution of the *multiply-and-accumulate* (MAC) operations within the *fully connected* (FC) and *convolutional* (CONV) *layers* that compose the DNN model and to which a memory system is connected. The later is usually split into on-chip and off-chip storage elements that hold the network parameters of all the computational units of the implemented DNN model. However, depending on the selected hardware platform, this architectural organization undergoes variations, adapting itself more or less well to the DL algorithm to be executed. Therefore, for the realization of the ANN inference accelerators, of fundamental importance turns out to be the choice of the processing unit architecturally better suited for DNN computation, to minimize the latency and energy consumption, and especially of the memory accesses, which often represent the main DNN processing bottleneck.

In this scenario, all the possible processing units capable of meeting the latency and energy consumption constrains imposed by edge computing represent a possible solution for DL inference processing at the edge. These include general-purpose processors (GPPs), such as Central Processing Units (CPUs), Graphical Processing Units (GPUs), re-configurable hardware platforms, such as Field Programmable Gate Arrays (FPGAs), and special-purpose Application Specific Integrated Circuits (ASICs) designed specifically for ML applications, Tensor Processing Units (TPUs). Each solution has specific features in terms of flexibility, speed and energy efficiency,

where each solution has different characteristics in terms of flexibility, speed and energy efficiency. The goal is to find the best compromise between these three metrics for the ML application under consideration, considering that usually higher flexibility results in lower efficiency, and vice-versa.

While CPUs are the most flexible, they also have the worst performance in terms of processing speed and power consumption. TPUs are optimized for fast DNN processing with low energy, at the cost of inflexibility. GPUs can provide flexible computing power and high result accuracy at the cost of high energy consumption which likely prevent their integration in embedded devices for inference processing at the edge. FPGAs often provide the best compromise, providing high level of parallel and pipelined computation with very low energy consumption which are well suited to process the stream of calculation within DNN models (albeit at lower clock frequencies). In addition, the high degree of architectural flexibility and re-configurability of FPGAs enables the realization of custom logic units tailored to the ML application under consideration and supports the application of a wide range of optimizations techniques capable of optimizing the hardware accelerator metrics in terms of architectural improvements and in terms of DNN model simplifications, through the development of optimization algorithms at hardware/software level.

For these reasons, FPGAs became the optimal hardware platform, in terms of speed and energy efficiency, for complex DNN model hardware implementations for acceleration at the edge.

However, the increase of computational and storage complexity of DL algorithms can be a problem for running ML applications on small FPGAs, with very limited hardware resources. In particular, the increase in the number of network parameters of the current DNN models has not only determined the possibility to obtain a better accuracy in comparison to the standard ANNs, but it has also introduced a high redundancy in their values necessary to make easier and more precise the training process. This has translated in an increase of the range of the possible architectural solutions with which it is possible to obtain the same result for the task in question, also applying minimal simplifications to the considered DNN model.

Specifically, the increased size and complexity of current DNN models has not only determined the possibility to achieve better accuracy compared to standard ANNs, but also requires a high redundancy in the network parameters and increased the range of the possible solutions that can obtain the same result for the task at hand, even applying minimal variations to the considered computational structure.

Thus, the FPGA high flexibility enabled the implementation of DNN models on optimized architectures through the development of highly parallel computing paradigms capable of supporting energy efficient data flows. In particular, FPGAs support the development of the spatial architecture paradigm, which divides the on-chip memory into a hierarchy of different levels allows for maximizing the data reuse at the low levels. Since the access to the off-chip memory is the main bottleneck in DNN processing in terms of both latency and energy consumption, it reduces the latency and the energy consumption of the FPGA-based DNN inference accelerator significantly. In addition, for ML applications that tolerate a certain loss of accuracy, the FPGAs support for a wide range of approximating computation techniques capable of reducing the computational and memory requirements through a reduction of the DNN model over-parameterization, becomes of paramount importance for transforming dense DNN models into sparse versions with reduced precision. This reduction in model complexity translates in a decrease from both the computational and memory side, resulting in more efficient DL algorithm implementation for DNN inference processing at the edge.

An additional confirmation of the efficiency of the solutions based on FPGAs for the acceleration of DNN inference on the edge regarding the alternatives can be obtained by analyzing the so-called *roofline model*, which outlines the limiting factors for the throughput of a hardware accelerator as the number of available PEs and the bandwidth of the off-chip memory with respect to selected processing unit, to the maximum achievable performance level. In particular, it shows

that FPGA-based DNN inference accelerators custom designed for the ML application under consideration capable of supporting the application of approximate computational methods outperform all alternative solutions both in terms of performance, provided that some loss of accuracy on the produced results is accepted.

This thesis work explores solutions to the problem of accelerating the DL inference process at the edge through hardware implementation of increasingly complex DNN models on FPGAs, evaluating advantages and disadvantages introduced by the application of a wide range of optimization techniques applicable both at architectural and hardware/software level. Although there are many optimization methods working at the hardware/software level, the focus of this work has been mainly on the study of the so-called *approximating computation methods*, which have shown the most promise in terms of improving speed and energy efficiency for ML applications where a certain amount of degradation in accuracy is acceptable. The right combination of such optimization algorithms allows to reduce the complexity of DNN models making its implementation feasible on embedded devices characterized by a limited amount of resources and low energy budget. As a result, they become of paramount importance for FPGA-based DNN accelerators for inference processing at the edge.

Specifically, at the architectural design level the goal is to implement highly parallel computational paradigms with energy efficient data flows that best match the computational stream within DNN models by exploiting the spatial architecture on which is based FPGAs, with the aim of maximizing the data reuse of all types of data possible within the DNN model layers at the lower level of the on-chip memory hierarchy. Reducing the number accesses to off-chip memory allows to achieve significant improvements in terms of speed and energy efficiency for the FPGA-based DNN inference accelerator designed.

With regard to the optimizations at the hardware/software level, the idea is to apply optimization algorithms capable of reducing the computational and memory requirements per DNN inference iteration by reducing the model complexity thorough two classes of optimization techniques, which are those based on the reduction of the operand precision through different types of *data quantization* methods, and those based on the reduction in the number of operations per inference iteration and the model size through *model compression*, capable of increasing the level of sparsity of DNN model.

For the first category of methods, this result is achieved both reducing the computational precision of the weights and activations processed by MACs moving from complex towards simpler data types with lower bit widths, to achieve representations that require less hardware resources for DNN model storage and processing. It has been shown that the capability of FPGA to manage variable bit widths depending on the considered data and layer type of the DNN model results in improvements in both speed of execution of MACs and utilization of the on-chip memory, allowing to increase the speed and energy efficiency of the whole system. In addition, when the tolerance on accuracy loss may be large, extreme quantization techniques allows to significantly reduce the hardware utilization for processing units in terms of memory and computational requirements, resulting in highly efficient execution in terms of speed and energy consumption.

For the second class instead, this reduction is achieved by eliminating the MACs which do not influence too much the accuracy of the results produced compared to that that of the original DL algorithm. Regarding the reduction of network parameters, optimization techniques like *network pruning* and the *exploitation of ReLU statistics* allow increasing the level of sparsity of the model by setting to zero the values of weights and activations. Then, the introduction of *non-zero skipping* methods allow to avoid the execution of MACs involving multiplication with at least one null operand, drastically reducing the computation demand for DNN inference iterations. While alone this translates in a smaller number of computation for iteration of inference, if combined with techniques of compression of the sparsity it translates in a high saving of memory requirements. In addition, in view of the reduction of the model size, it is possible to either exploit methods

that allow to obtain accuracy levels of complex networks on simpler networks, such as *knowledge distillation*, or the development of compact networks, such as *SqueezeNet* and *MobileNet* DNN, able to reduce the number of network parameters in the design phase without with a limited reduction of the final accuracy of the model.

To summarize, through these approximating methods based on the idea to trade-off limited accuracy it is possible to achieve a significant reduction in both computational and memory resource consumption for the implementation of the DNN models on resource-constrained device, resulting in a substantial performance improvement in terms of speed and energy efficiency. Specifically, the more the DNN model structure is approximated, less hardware resources will be required to implement the DL algorithm, for storing weights and activations to be processed and for performing the MACs required for each DNN inference iteration.

The high level of programmability and reconfigurability of FPGAs makes them the optimal processing unit terms of speed, energy efficiency, and hardware resource consumption for performing DL inference at the edge, for the development of DNN inference accelerators capable of both handling internal parallelism over a variable number of bits and exploiting the sparsity of the DNN model introduced by data quantization and model compression methods, respectively.

This analysis focuses on hardware implementation of ANN types for which FPGAs are the most promising processing units for accelerating the DNN inference processing at the edge, which specifically are deep *MultiLayer Perceptrons* (MLPs) and Convolutional Neural Networks (CNNs) based on the traditional artificial neuron model. They focus on maximizing the model performance and accuracy, because of their highly parallelizable feed-forward execution flow in a pipelined fashion. In addition, alternative ANN architectures like SNNs use asynchronous internal computations and are inherently sparse, which make them very interesting in the context of developing model DNNs on FPGAs for low-power real-time ML applications.

Contents

List of Figures

List of Tables

Introduction

In recent years, with the exponential increase in the amount of data available, *Deep Neural Networks* (DNNs) have become the most widely used computational model in solving complex problems for most of applications, due to the extremely high level of accuracy they are capable of attaining, compared to that of traditional *Machine Learning* (ML) algorithms.

However, to achieve this value of precision on the results produced by the system, requires an increasing in the number of layers and neurons within the *Neural Network* (NN) architectures, resulting in a huge demand of computational and storage resources to handle the increasing number of *multiply-and-accumulate* (MAC) operations and parameters, as well as high memory bandwidth to maximize computing performance.

Because of this increase in computational intensity and storage resource consumption for both *Deep Learning* (DL) algorithm training and inference processing, the implementation of DNNs on traditional *General Purpose Processors* (GPPs) based on the sequential execution of the corresponding instruction stream on a fixed architecture, known as Von Neumann, becomes a problem for ML applications that require a certain level of performance in terms of latency and energy consumption.

To address this challenge, the different types of inherent parallelism exhibited by DNN models has motivated over time the development of dedicated *hardware accelerators* capable of exploiting the internal concurrency in the execution of MACs within *Convolutional* (CONV) *layers* and *Fully Connected* (FC) *layers* required for DNN processing, where the choice of the optimal processing unit depends on the DL algorithm phase to be processed.

Specifically, as the computational complexity and memory requirements of training are much higher than those of inference, the optimal choice for DNN processing is to perform the former on the cloud and the latter at the edge. Thus, while the training on the cloud is usually processed in high floating point precision on computational powerful and high memory band-width *Graphics Processing Units* (GPUs), with the aim of improving the speed and energy efficiency, it would be preferable to perform the inference directly at the edge, thus avoiding the data transmission latency and energy cost required for moving DNN processing from the edge to the cloud.

In the field of DL for edge computing, hardware DNN inference accelerators custom designed for the ML application under consideration based on processing units with re-configurable and flexible architecture are resulted the most promising solution to improve the implementation of complex DNN models on low-power/low-latency and resource-constrained embedded devices, due to their small size, high speed and energy efficiency. However, the limited amount of computational and storage resources, coupled with the low off-chip memory bandwidth of both *Application Specific Integrated Circuits* (ASICs) and *Field Programmable Gate Arrays* (FPGAs) limit the performance in terms of speed and energy efficiency of such type of hardware DNN inference accelerators, restricting their scope of application.

Although the implementation of complex DNN models on ASICs allows for the best performance in terms of both throughput/latency and power consumption, the total lack of reprogramming and the high design cost restrict their use. Alternatively, FPGAs, due to their high degree

of re-programmability and architectural flexibility in terms of designing specific logical units, represent the best compromise for DNN inference processing at the edge to improve speed by maximizing the exploitation of the inherent parallelism of DL algorithms and to achieve high energy efficiency through the application of some optmization techniques.

Specifically, while system throughput is mainly limited by both the number of *Processing Elements* (PEs) available on-chip dedicated to the execution of MACs for DNN inference processing and the off-chip memory band-width containing the DNN model to be evaluated, the most significant energy consumption contribution is due to the corresponding memory accesses and data movements required to process the data per DNN inference iteration.

This thesis work addresses the problem of accelerating the DL inference process at the edge through hardware implementation on FPGAs of increasingly complex DNN models. More in detail, FPGA-based DNN inference accelerators that are speed- and energy-efficient can be implemented through the development of optimization techniques that work at different levels. Specifically, while at the *architectural design level* the goal is to implement highly parallel computational paradigms with energy efficient data flows that best match the computational stream within DNN models, at the *hardware/software level* the idea is to apply optimization algorithms capable of reducing the computational requirements and memory demand in terms of size and off-bandwidth, maximizing the reuse of data at the low level of the memory hierarchy, while keeping the accuracy of the results produced as close as possible to that of the original DL algorithm.

Among these, the ones that have been proven to be the most effective for DNN inference processing on FPGA are the so-called *hardware-oriented optimization techniques for approximate computation*, which allow improve the speed and energy efficiency of a FPGA-based DNN inference accelerator by reducing the complexity of the implemented DNN model, at the expense of some loss of accuracy. These approximating methods follow two directions, which are the reduction of the computational precision of the weight and activation to be processed through *data quantization* and the reduction of the number of MAC operation and model size through *model compression*. Specifically, the more the DNN model structure can be approximated, the fewer hardware resources will be consumed in the implementation of the DL algorithm for storing the parameters to be processed and for performing the calculations required per DNN inference iteration, leading to an improvement in both speed and energy consumption. Hence, the idea is to trade-off a a certain quantity of accuracy due to approximation of computations performed in DNN inference processing to achieve both a significant reduction in hardware resource consumption for the implementation on FPGA of the DNN model under consideration, and a substantial performance improvement in terms of speed and energy efficiency.

All the techniques for optimizing the development of FPGA-based DNN inference accelerators analyzed are applicable to the implementation of a wide range of ANN types. The analysis carried out focuses mainly on the three types of DNN models, for which FPGAs have emerged as the most promising processing units to accelerate DNN inference processing at the edge, which are deep MLPs and CNNs, which focus on maximizing the performance and accuracy of the model, and SNNs, which are mainly used for low-power ML applications. This is due to the feed-foward nature of such types of ANNs allows maximizing the exploitation of the inherent parallelism of DNN models at various levels following a pipelined hardware execution available on FPGAs.

Chapter 1

State of the Art in Artificial Intelligence and Machine Learning

In the following sections a general background on the basic concepts related to the *Deep Learning* (DL) field of study in the area of *Machine Learning* (ML) will be introduced, making a distinction between the various possible types of *Artificial Neural Networks* (ANNs), and trying to provide a general description of the main limitations encountered in the implementation of *Deep Neural Networks* (DNNs) on embedded systems, and possible solutions.

1.1 From Artificial Intelligence to Deep Learning

It is important to understand the differences between Artificial Intelligence, Machine Learning and Neural Networks (see Figure 1.1).

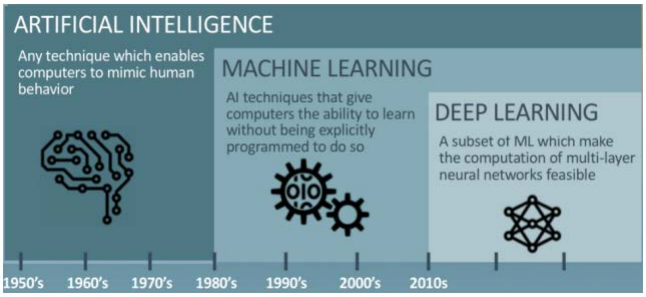

Figure 1.1: Evolution of Artificial Intelligence. Source [1]

Artificial Intelligence (AI) is the science that studies the ways to build systems able to solve complex problems like humans do.

A subset of the AI area is the *Machine Learning* (ML) field, allowing the systems to automatically perform a task without being explicitly programmed, and so, with this type of systems, it is possible to solve problems thanks to the experience acquired during operation. Thus, ML is a subset of AI that defines models able to learn from and make decision on *features* extracted by a set of data, without being programmed for specific tasks. Within the ML field of study, there is an area referred to as *brain-inspired computation*, in which algorithms emulate how the human brain works.

The brain-inspired computing field is in turn divided in two sub-areas, called respectively

Artificial Neural Networks (ANNs) and *Spiking Neural Networks* (SNNs), which are computational models used to represent biological neural networks more or less faithfully, according to the required level of detail. The main difference among the two is the time model, continuous for *ANNs* and discrete for *SNNs*.

Within ANNs, *Deep Learning* (DL) is based on *Deep Neural Network* (DNN) architectures, which can improve the accuracy at the expense of larger amounts of data to be analyzed and increasing computational complexity. The higher DL performance over ML techniques comes from the DNN ability to automatically learn features from the input data.

The relationship among ML, ANNs, DL and SNNs is illustrated in Figure 1.2.

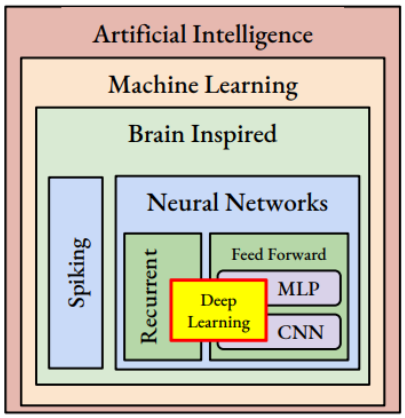

Figure 1.2: Artificial Neural Networks, Deep Learning and Spiking Neural Networks as sub-fields in the context of Machine Learning. Source [2]

Figure 1.3 shows a comparison in terms of achievable model accuracy performance between DNNs and alternative approaches. It is clear that as the amount of data to be processed increases, the choice of DL algorithms is the best possible to obtain the highest level of performance on the results produced.

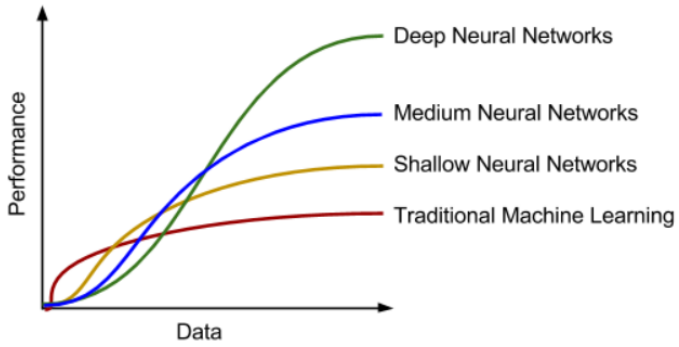

Figure 1.3: Performance comparison of Deep Neural Networks versus alternative approaches as a function of the amount of data to be analyzed. Source [3]

1.2 Artificial Neural Networks and Deep Learning Overview

The idea behind the ANN structure has been inspired by biological neural systems, consisting of interconnected *neurons*, which constitute the basic computational units (often called *nodes* or *units*), that communicate through electrical pulses.

Each biological neuron (see Figure 1.4, on the left) consists of several components: a *cell body*, which includes a *nucleus*, where the information is processed, multiple *dendrites*, by which it receives the input signals, a single *axon*, along which it produces the output signal, and multiple *synapses*, consisting in connections between two neurons, called *presynaptic neuron* and *postsynaptic neuron*, responsible to transmit the information towards the dendrites of other cells.

The behaviour of such structures is based on the fact that only when the input voltage exceeds a certain threshold, an impulse is generated at the output, called *action potential*, otherwise no response is produced. Any synapses can modify the value of the signal crossing it, and the way the brain learns is through changes in the value associated to them.

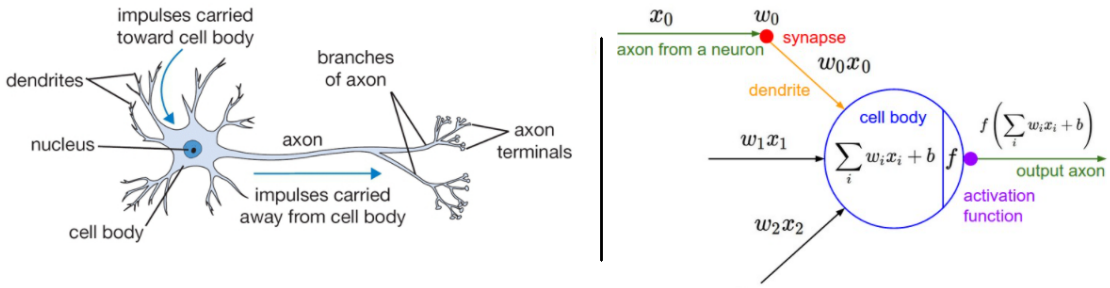

Figure 1.4: Biological (on the left) and mathematical model (on the right) of a neuron. Source [4]

Hence, each neuron receives some inputs (also referred to as *input activations*) from other neurons, and then it computes a single output (also referred to as *output activation*). Inspired by this idea, as shown in Figure 1.4 (on the right), it is possible to obtain a mathematical model of a neuron, as a node with multiple inputs, where at each of these connections is associated a certain value (referred to as *weight*), that multiplied to the correspondent input signal value, allow to compute the output signal of the unit in exam. From a mathematical point of view, each node of a NN performs a specific operation represented by a function of the weighted sum of all its inputs

$$y = f\left(\sum(w_i x_i + b)\right),$$

where y represents the *output activation*, w_i the weight associated to the ith connection, x_i the ith *input activation* associated to the considered node, b the *bias* parameter, and $f(.)$ is the non-linear *activation function*. Thus, the multiple inputs received by a neuron are multiplied with the weights, summed together with the bias term, a then on the result is applied a non linear function, which allows to generate an output only if the signal crosses a certain threshold value.

The weights and activations are commonly organized by layers, in matrices and vectors respectively, referring to every weighted sum operation as a *matrix–vector multiplication*.

Figure 1.5 shows the most commonly used non-linear activation functions.

The use of non-linear activation functions allows the introduction of some non-linearity into the NN computational model. *ReLU* represents the most commonly used non-linear activation function in DNN models, due to its ability to increase their level of *sparsity*, so the number of zero network parameters.

A structure composed of just one single neuron implies some limitations regarding the executable tasks. The solution is to realize an architecture referred to as *Artificial Neural Network*

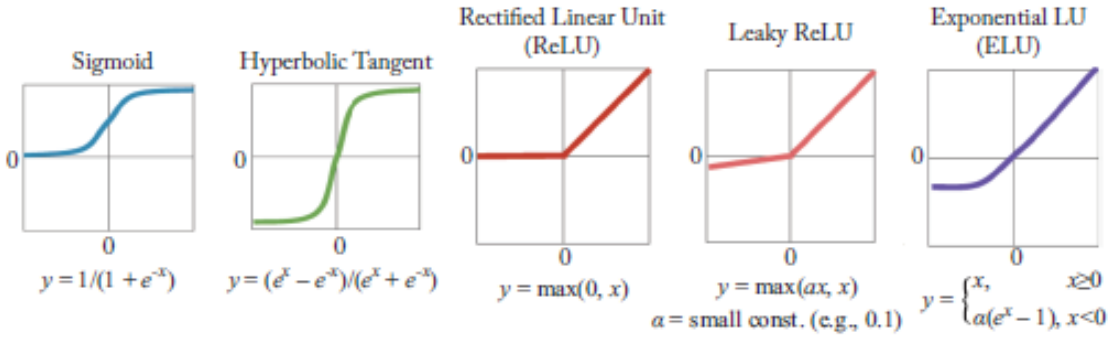

Figure 1.5: Most commonly used non-linear activation functions. Source [5]

(ANN) [or simply to as *Neural Network* (NN)], structured as multiple blocks of neurons, organized in the so called *layers* (see Figure 1.6), where each unit receives its inputs and sends its output to other nodes.

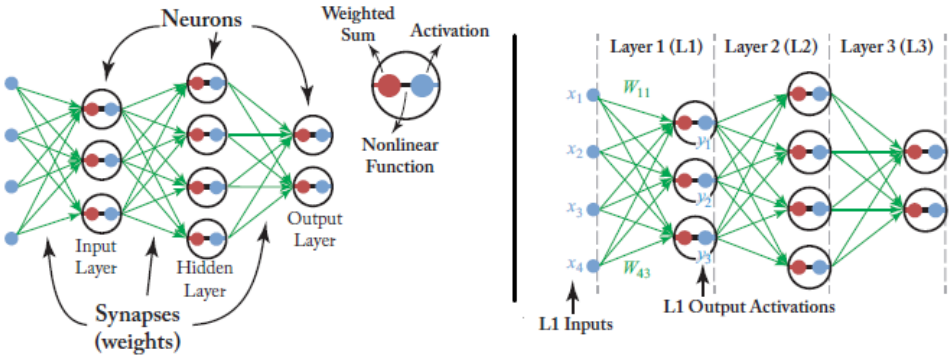

Figure 1.6: Neural Network structure (on the left) and definitions (on the right). Source [6]

Any ANN model is composed of three different types of layers. The *input layer* consists in a set of inputs nodes that receive the data in input to the network and pass information to the next level layer, called *hidden layer*, where it is executed the computation of the weighted sum operation. The number of hidden layers present in an ANN determines the depth of the structure under examination. Finally, the results obtained from hidden layers are propagated to the *output layer*, responsible to provide the final outputs of the network.

The depth of the network is strictly related to the complexity of the task, and ANNs with more hidden layers are often referred to as *Deep Neural Networks* (DNNs). Figure 1.7 compares two examples of possible architectures for ANN and DNN models.

More in detail, deeper ANNs can implement more complex functions and thus can achieve higher accuracy. However, the improvement in performance of DNNs compared to ANNs is obtained at the cost of an higher computational complexity.

In order to provide a general overview about ANNs, an important aspect to take into account is the one related to the network topology.

We can identify two main forms of NNs, which are *Feedforward Neural Network* (FNN) and *Recurrent Neural Network* (RNN). In the former, all the computations needed to calculate the output of the network is performed as a sequence of operations in the forward direction. The most common known example of feedforward DNNs are the *MultiLayer Perceptrons* (MLPs).

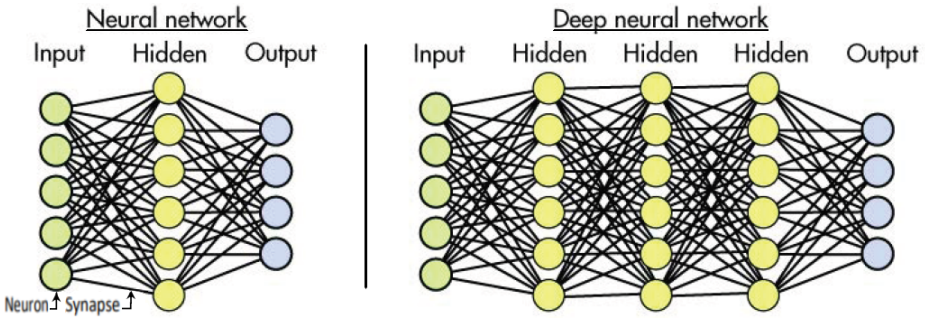

Figure 1.7: Neural Network (on the left) versus Deep Neural Network (on the right). Source [7]

In contrast, the latter is a type of DNNs where the sequence of operations is performed also in the backward direction. Figure 1.8 shows an illustration of the two different types of ANN forms.

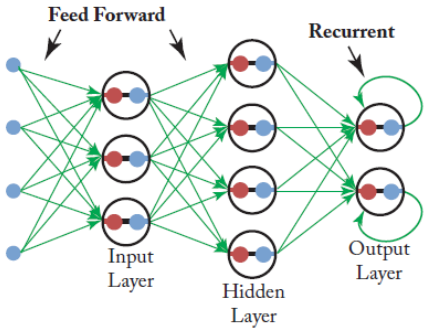

Figure 1.8: Different types of Artificial Neural Networks: Feedforward Neural Network versus Recurrent Neural Network. Source [6]

1.3 Inference and Training Process Considerations

The DL learning process consists to modify the value of the network weights and biases, processing the input data, and it is referred to as *training phase*. Once DNN is trained, it is able to solve the task in exam by computing its output, and it is referred to as *inference process*. The other type of variables of the network used to control the learning process are the so-called *hyper-parameters*, which are not estimated during training, but they need to be set by the user before starting the learning phase, and then manually adjusted iteratively.

One of the most commonly used learning paradigm is the so called *gradient descent backpropagation*. It is a repeated iterative process of propagation through the network, where the propagation of the inputs to the output is called *forward-pass*, while the propagation of the error from the output to the input computed evaluated using the *loss function*, which in turn provides a measure on how the predicted output is near to the expected result, is called *backward-pass*.

Then the computed loss value L is used during the *backpropagation algorithm* execution, which compute the gradient of L for each weight and bias within the ANN. Thus, to perform the learning process, it adjusts each network parameter by an amount proportional to the partial derivative of L with respect to network parameters through some optimization algorithms, reducing the loss function value to minimize the network prediction error. One of the most common optimization algorithm is the so-called *gradient descent*.

The goal of training is to adjust weights and biases to minimize the value of the loss function as much as possible, in order to both to maximize the model *accuracy* on the results produced and improve the *generalization* in order to make correct predictions on data different from those used in the training.

During the training, the *overfitting* problem occurs when the NN to train is too complex, thus having a too high number of parameters to update. When it happens, the DNN model fits the training data too accurately, memorizing the associated correct result than learning to generalize over the new data used during the inference. A possible solution to the overfitting problem is to switch to a less complex DNN model that performs the same task with a lower accuracy.

To make a distinction among all *learning paradigms* to train an ANN, it is possible to classify them in three main types. Specifically, the first one is the *supervised*, when all the training samples are labelled, and the learning algorithm determines the output from the input data and its corresponding labels. In *unsupervised learning* the training samples are not labelled, and the network is trained to find patterns in the input data. *Semi-supervised learning* is a trade-off among the two. An alternative approach to determine the network parameters is *transfer learning*, where the weights and biases of an already trained model are transferred to another untrained one. Thus, through its application it is possible to set a point for the untrained model for starting to adjust the parameters for executing a new task (*fine-tuning*), making the training process faster than when it starts with random parameter values. By fine-tuning a DNN model, it is possible to recover some amount of accuracy lost for example to the application of some approximating method to *reduce precision* or to *reduce the number of operation* or the *model size*.

It is important to highlight a couple of points about DL algorithm processing. First, the training requires much more computational and storage resources compared to the inference, due to the fact that the inference requires only a forward-pass of the network, while in training, both the forward-pass and the backward-pass are performed multiple times. Second, the needed precision for the training is much higher compared to the inference one, and so in such case it is not possible to apply many approximate computation techniques to reduce the computational and storage demand.

For these reasons, usually the former is performed on the cloud, where the amount computational power and hardware resources is very high, while the latter is executed at the edge for avoiding the latency and energy consumption for data and model transfer from embedded device to the cloud. Consequently, since on embedded systems the amount of resources is limited and the energy budget is a critical factor, the development of techniques able to reduce the computational cost for the DNN inference processing at the edge has become of paramount importance.

1.4 Types of Layers in DNNs and CNNs

The architecture of DNNs is composed of many layers of different types, which differ from each other depending on how the connections between layers are made. If in a layer all the output are composed of a weighted sum of all inputs, it is referred to as *fully connected* (FC) *layer*, while if in a layer some of the outputs depend just on a subset of the inputs, then it is referred to as *sparsely connected* (SC) *layer* (see Figure 1.9).

Thus, being that the total number of parameters associated to a FC layer directly depends on the number of its inputs/outputs, its implementation on hardware platforms with a limited amount of memory and computational resources can be a challenging task.

To solve this problem, it has been introduced a new type of layer, referred to as *convolutional* (CONV) *layers*.

Convolutional Neural Networks (CNNs) are a kind of DNNs able to reduce the number of parameters per layer, adopted when the data in input to be processed have a *spatial* or *temporal relation* to each other. The idea here is to exploit the existing correlation between inputs close

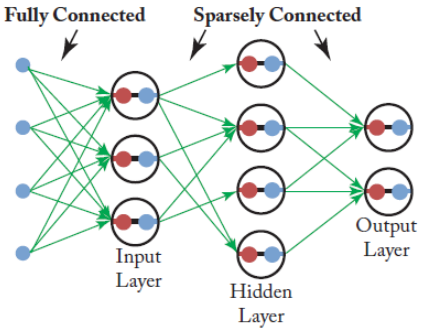

Figure 1.9: Different types of connections in Artificial Neural Networks: Fully Connected layer versus Sparsely Connected layer. Source [6]

to each other to reduce the number of layer parameters. The main difference to the traditional ANNs is the introduction a new type of layer, referred to as *convolutional* (CONV) *layer*.

It is based on the convolution operation, which makes it possible to both significantly reduce the amount of layer parameters, exploiting the *weight sharing* property, consisting in the repeated use of the same set of weights to be multiplied with the input activations for the computation of different output activations.Thus, the structure of any CNN is mainly composed of a sequence of multiple CONV layers ending with a FC layer, as shown in Figure 1.10.

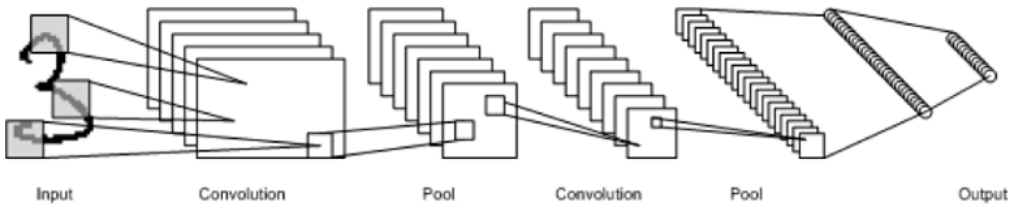

Figure 1.10: Basic layers of a typical Convolutional Neural Network. Source [8]

More in detail, a standard CNN architecture is composed of many different types of layers which are:

– *Convolutional* (CONV) *layer*: it is the layer performing the convolution operation (see Figure 1.11) between the inputs and a structure referred to as *filter* or *kernel*, capable of extracting information from them. In this calculation, both input and output relative to a CONV layer are organized as a set of matrices referred to as *feature maps*, each of which is indicated as *channel*. Kernels contain filter weights that applied sequentially by sliding them on the input feature map generating the corresponding output future map. Thus, the weighted sum between weights and activations for each output is calculated using only a limited set of inputs per time, denoted as *receptive field*.

– *Non-linearity layer*: it is the layer applied to introduce some non-linearity (see Figure 1.5) in the CNN computation. It allows both to remove useless extracted information, and to improve the level of sparsity of the DNN model.

– *Pooling* (POOL) *layer*: it is the layer inserted after each CONV layer to both reduce the size of the data resulting from the convolution operation and speed-up the computation.

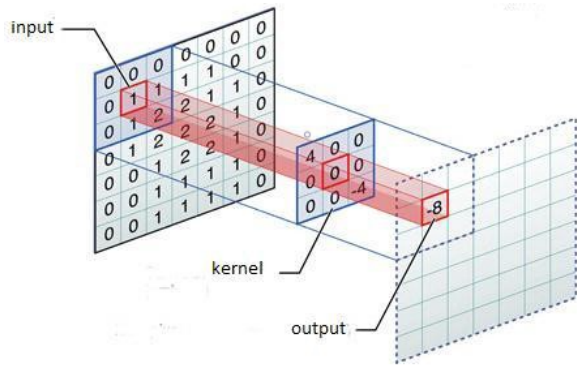

Figure 1.11: Representation of a convolution operation performed in CONV layers. Source [9]

The *average* and the *max pooling* (see Figure 1.12) are the most common kinds of pooling operations.

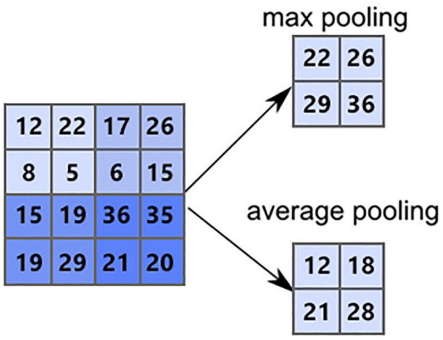

Figure 1.12: Numerical example of average and max pooling operations. Source [10]

– *Fully connected* (FC) *layer*: it is the layer responsible to provide the final output processing data contained in the final feature map.

– *Normalization* (NORM) *layer*: it is an optional layer that can be inserted to speed up the training. Specifically, since the distribution of input data values can be variable, it scales these values to have zero mean and unit variance, in order to avoid overflow/underflow in the internal DNN calculation, that would result in an accuracy loss.

Thus, the standard architecture of a CNN is composed of a combination of two basic building blocks, which are the *convolution block*, consisting of a CONV layer and a POOL layer, responsible for extracting useful information from the input data and progressively reducing its data size, and the *fully connected block*, consisting of a FC layer, responsible for producing the final result.

Thus, it is possible to say that a complete CNN architecture consists of a sequence of CONV blocks, followed by a single FC block.

Input → CONV + ReLU → POOL → ... → FC

Over the years, using different sequences of these blocks, it has been possible to build up several architectures for CNN models. Some examples includes LeNet, AlexNet, VGGNet, GoogLeNet and ResNet.

It is possible to see that while the combination of used building blocks type is always the same, it has been a progressively increasing in their number, and thus in the amount of network

parameters and operations performed in each CNN model.The idea is that the more the level of accuracy to be achieved, the larger is the number of layers, and so of parameters and operations required, and consequently it implies an higher energy consumption and lower throughput/higher latency.

It is possible to make a distinction between the different CNN models in terms of the achievable *top-n error rate*[1], where each one requires a different number of parameters and operations to be obtained with a specific accuracy value.

To have an idea about the relation between DNN model complexity and achievable accuracy, Figure 1.13 shows the dependence of the *top-1 accuracy* on the number of operations and parameters required per inference iteration of the standard CNN models.

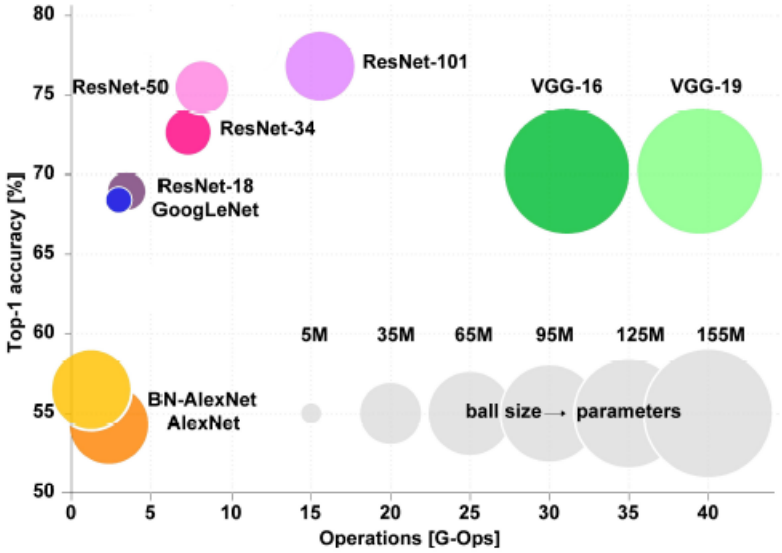

Figure 1.13: Top-1 accuracy versus number of MAC operations required per DNN inference iteration. (The size of the circles is proportional to the number of parameters). Source [11]

Figure 1.14 shows the accuracy measured as the top-5 error rate of different DNN models from *ImageNet Challenge*.

As a result, compared to ANNs, DNNs require a significant increase in the number of parameters and operations to be performed for processing the same task, but allowing to achieve a higher level of accuracy. Both CONV and FC layers internal computations are based *multiply-and-accumulate* (MAC) operation, since both involve the matrix-vector multiplication operation between weights and activations. In Table 1.1 have been reported the amount of parameters and MACs for some popular CNN models in both the FC and CONV layers. It is possible to state that all MACs and parameters required for DNN processing take place in the CONV and FC layers.

The first thing that can be noticed is that the computational (for operations) and memory (for parameters) requirements of DNNs is high, and since the DL algorithm complexity increases with the size of the input data to be processed, it results in an additional increasing in the number

[1]The *top-n error rate* is measured based on whether the correct model result appears in one of the top-n output values produced. The most common evaluating metrics to evaluate the model accuracy are top-5 and top-1 accuracy.

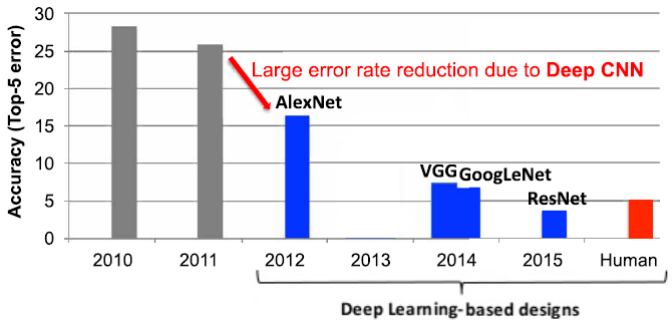

Figure 1.14: Comparison of DNNs in terms of top-5 accuracy in ImageNet Challenge. Source [6]

of operations and parameters needed, leading to a further increase in computational and storage complexity for DNN models.

Table 1.1: Comparison of parameters and MAC between different CNN models. Source [6]

	LeNet	*AlexNet*	*VGG*	*GoogLeNet*	*ResNet*
CONV Parameters (weights/biases)	2.6 k	2.3 M	14.7 M	6 M	23.5 M
CONV Operations (MACs)	283 k	666 M	15.3 G	1.43 G	3.86 G
FC Parameters (weights/biases)	58 k	58.6 M	124 M	1 M	2 M
FC Operations (MACs)	58 k	58.6 M	124 M	1 M	2 M
Top-1 Accuracy	49.5 %	61 %	74.5 %	77 %	79.3 %

In edge computing, usually characterized by stringent power consumption and low latency constraints, a too complex DNN precludes its implementation on embedded devices, because the larger the model size, the higher the number of MACs and parameters required for its processing. As a result, over time a wide range of architectural improvements and optimization techniques able to reduce the increasing demand of computational and storage requirements have been developed to make its implementation feasible on resource-constrained systems.

Looking once again at Table 1.1, a couple of points deserve to be highlighted. First, the number of parameters needed in the FC layers is always higher than in the CONV layers, thus requiring higher memory requirements in terms of dimension. Second, the CONV layers are where most of the MACs are performed. Second, the CONV layers are where most of the computation is performed.

To have an idea, Figure 1.15 shows the distribution of parameters and operations in the case of VGGNet, which is very similar to the one of other types of DNNs.

Thus, the CONV and FC layers combined contribute to more than 90 % of the network parameters and MAC operations, while the other types of layers of DNNs contribute little to the computational and storage requirements. As a result, in the hardware implementation of current DNN models, most of the optimization techniques are focused accelerating the processing of these two types of layers, not considering the others.

In addition, the system energy cost is mainly due to the memory data transfers than the computational part. Specifically, the largest contribution is caused by the CONV layers, being them the most computationally intensive, requiring more memory accesses, while the remaining part is attributable to FC layers.

So even to optimize DNNs implementation in terms of energy efficiency, the main effort must focus on these two kinds of layers. As the current trend is to make deeper DNN models by increasing the number of CONV layers while maintaining the same amount of FC layers, it is

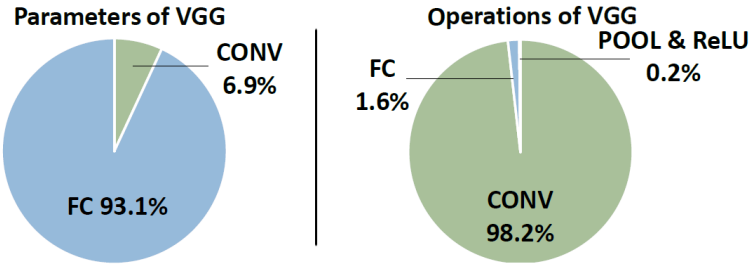

Figure 1.15: Operation and parameter requirements for VGGNet fully connected (on the left) and convolutional layers (on the right). Source [12]

required a particular attention on their energy cost optimization.

1.5 An Introduction to DNN Efficient Embedded Implementations

The implementation of increasingly complex DL algorithms capable of achieving extremely high levels of accuracy on resource-constrained embedded systems, where the main objective is to achieve a high throughput/low latency and small power consumption is a challenging task. More in detail, the need to perform within DNN models a large amount of MACs in parallel does not fit well with the sequential execution of the DL algorithm software instructions on processors based on the Von Neumann architecture (see Figure 1.16).

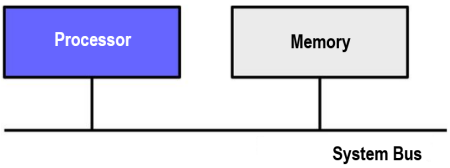

Figure 1.16: Block diagram representation of Von Neumann architecture. Source [13]

Specifically, the main problems are in the separation between processor and memory containing the data to be processed, and in the fixed architecture of the processor that can not be adapted to the DL algorithm at hand. Thus, the high level of DNN model parallelism exhibited by DNN models has motivated the developed of dedicated *hardware accelerators*, capable of achieving high performance through a high degree of parallel computation, while maintaining an high energy efficiency.

As mentioned before, although the type of computations to be performed is exactly the same for both inference and training, the latter is much more expensive than the former, resulting in about twice the computational and memory cost. For these reasons, there is no reason to perform training directly on edge devices. Thus, DNNs are trained only once off the chip, and then the resulting trained model is loaded onto the hardware platform chosen to accelerate the inference. As a result, since the training is usually performed off-line in the cloud, for its execution there is the need of highly parallel *Graphics Processing Units* (GPUs) optimized for the high precision floating-point computation, capable of decreasing the training time performing many MACs simultaneously, without having to worry about the computational and energy budget constraints.

However, for performing on-line the inference at the edge, it is necessary to find solutions to make possible the implementation of deeper and deeper trained DNN models on an embedded

device. It allows for avoiding to send the DNN model and data to be processed to the cloud, improving both the latency and power consumption of the processing. To accelerate the DNN inference processing on embedded devices, it is possible to exploit the different types of parallelism exhibited by DNN models implementing DL algorithm in hardware.

More in detail, at the level of neurons in the same layer, it is possible to execute multiple MACs simultaneously, as there are no *data dependencies* between the data to be processed, exploiting the so-called *neuron parallelism*. Then, at the level of layers, thanks to the feed-forward structure of DNNs composed of a sequence of dependent layers, it is possible to perform all calculations associated with each layer in a pipelined way, by taking advantage of the *layer parallelism*. Finally, at the level of inputs, it is possible to processes a set of incoming samples at a time (*input parallelism*).

For DNN inference processing acceleration, especially for low-power and low-latency ML applications on embedded systems, it is fundamental to select the optimal processing unit on which implement the DL algorithm to execute. All the hardware platforms capable of supporting DNN model implementation are organized as an array of *processing elements* (PEs), each of which is capable of performing a MAC operation. The analysis for choosing the optimal compromise among all the processing units for edge inference processing must be made by looking at three metrics, which are *flexibility*, *performance* (i.e. throughput/latency) and *energy efficiency* (see Figure 1.17).

Figure 1.17: Comparison of different hardware platforms for edge inference processing. Source [14]

Despite their high degree of flexibility, *General Purpose Processors* GPPs are not widely used for DNN inference processing at the edge. While CPUs provide low speed and low energy efficiency, GPUs due to their high power consumption are not suitable for low-power applications, despite the high level of performance they can achieve in high precision floating point calculations.

Since in DNN inference the order and the type of operations to be performed are fixed, the programmability at the architectural level of custom hardware solutions such as ASICs and FPGAs allows for the best possible configuration of the hardware logical units tailored to the specific application. As a result, it enables the development of dedicated hardware architectures able to optimize the execution of the task in exam. Despite the better performance and energy efficiency of ASICs at the cost of low flexibility and high design cost, usually FPGAs are the best compromise between the three metrics listed above. In addition, due to their high level of re-configurability, FPGAs support the application of a wide variety of run-time optimization methods, which aim to improve both the speed and energy efficiency of DNN inference processing.

More in detail, it is possible to classify the optimization algorithms for accelerating DNN inference processing into two different classes, which are *hardware/software-level optimization techniques* and *architectural-level design improvements*, as shown in Figure 1.18:

- *Architectural-level design improvements*, which mainly focus on exploiting the characteristic of DNNs to adapt the internal architecture of the selected processing unit, so that DL algorithms can be implemented in hardware as efficiently as possible.
- *Hardware/software-level optimization techniques* which mainly focus on reducing the computational/storage requirements and memory bandwidth required to run DNN models in hardware, while attempting to affect the accuracy of the result produced as little as possible.

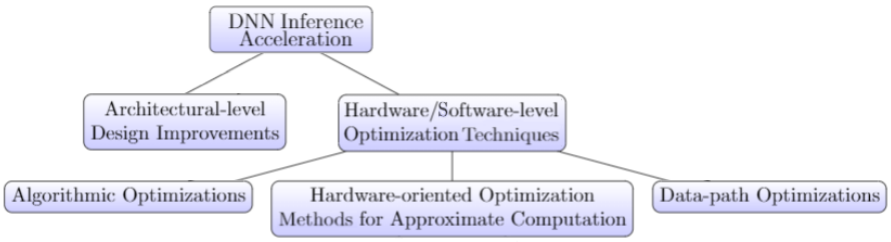

Figure 1.18: Architectural-level design improvements and hardware/software-level optimization methods to accelerate DNN inference processing.

The latter can be generally achieved in three main ways:

1. *Algorithmic optimization methods*, which exploit the features of DNN models through the application of computational transforms, in order to accelerate the calculations within the CONV and FC layers. The objective is to reduce the DL inference processing requirements on both computational/storage and memory bandwidth, and consequently, the DL algorithm complexity, without affecting the final accuracy. The best known examples of such computational transforms are the so-called *im2col method*, capable of mapping 3-D operations performed in CONV layers into matrix multiplications such as *General Matrix Multiplications* (GEMMs) [15], the *Fast Fourier Transform* (FFT) *method* [16], which transforms convolution operations into multiplications by shifting their execution to the frequency domain, and the *Winograd algorithm* [17], that allows to increase the throughout by simplifying convolution operations into multiplications by matrices.

2. *Hardware-oriented optimization methods for approximate computation* of DNN models, which in turn are divided into *reduced-precision computation* methods through *data quantization* and algorithms for *reducing the number of operations and model size* through *model compression*. They can trade-off a certain loss in accuracy for significant improvements in both speed and energy efficiency, resulting in a reduction of both computational and memory requirements. Data quantization involves reducing the precision of network parameters by changing the data representation of weights and activations from standard *32*-bit floating-point data type to a lower precision fixed-point one, depending on the required accuracy. More in detail, it allows both to reduce the memory bandwidth and storage size requirements, and consequently to increase the throughput and energy efficiency. Model compression consists in reducing the model size and number of operations involved in inference processing by making sparser the DNN model both at the level of connections between nodes and of number of neurons. Thus, the application of optimization algorithms belonging to these classes facilitates the hardware implementation of DNN models on resource-constrained devices.

3. *Data-path optimization methods* make structural changes to the DNN model architecture to take advantage of its inherent parallelism on the limited amount of computational resources. Specifically, the aim is to map onto a limited number of processing units the computations required for DNN inference processing by making some modifications on the flows of data moving in the internal architecture. The most common examples of such data-path optimizations are *loop unrolling* and *loop tiling*.

An important difference between the two main classes of optimization techniques mentioned above is that, unlike *architectural-level design improvements* that do not affect system accuracy at all, most *hardware/software-level optimization techniques*, improvements can significantly degrade the accuracy of the model.

Another aspect to be taken into account regarding DNN model implementations on embedded devices is the one related to memory accesses. Specifically, the high level of concurrency exhibited by DNNs can be exploited by allocating multiple PEs on the processing unit that, working in parallel, are able to accelerate the inference processing executing simultaneously as many MACs as there are PEs. However, to perform such operations, it is necessary to read/write data to be processed from/to memory. Since each MAC involves four memory accesses to be performed, also a small increment in the number of MACs required for DNN inference processing results in a huge increment of memory access demand.

Thus, for DNNs, the main limitation for DNN inference processing in terms speed and energy efficiency is not caused only by the computations, but by the memory accesses. It is due to the limited amount of memory size and bandwidth, which very often is not able contain all data per inference process iteration and to provide enough data to work all the PEs simultaneously, and the energy consumption to read the required stored parameters per inference iteration. In addition, usually the on-chip memory of the hardware accelerator is not sufficient to contain all the parameters required to perform a DNN inference iteration, and thus the presence of an off-chip memory is mandatory. More specifically, external memory accesses are more expensive in terms of latency and energy consumption than internal memory, greatly impacting on both the speed and energy efficiency of the system. Thus, the memory accesses to the external memory must be limited, if not avoided unless it can be done without.

As an example to prove that the storage part is much more energetically expensive than computation one, results reported in [18] and [19] show that a 32-bit floating point MAC consumes 3.7 pJ, while a single SRAM and DRAM access requires respectively 2× and 173× more energy, and thus approximately from 7 pJ to 650 pJ.

Thus, to optimize DNN inference processing on edge devices as both speed and energy efficiency, a special attention must be paid to memory accesses. At the architectural design level of the hardware accelerator, it is important to develop parallel computing paradigms capable of handling efficiently the execution of the MACs required for each inference iteration, and to reduce the number of off-chip memory accesses as much as possible, maximizing the reuse of data stored in the on-chip memory. For these reasons, the introduction of an on-chip memory hierarchy becomes fundamental, and thus the architecture of processing units on which base the development of the hardware accelerator must be capable of supporting its hardware implementation.

To summarize, it is possible to state that FPGA-based DNN inference accelerators overcomes largely the performance of GPPs for edge computing. In particular, FPGAs allow to realize an architecture-level design tailored on the specific DNN model to be implemented, able both to optimize computations and minimize the off-chip DRAM memory transfers, maximizing the reuse of the data stored at the low levels of the on-chip memory hierarchy. Moreover, the possibility to apply a wide range of approximation techniques helps to reduce the DNN model complexity in terms of precision of the data to be processed and amount of operations required for each DNN inference iteration, and thus to reduce also the number of off-chip memory accesses and the off-chip memory bandwidth demand.

Table 1.2 summarizes the key design goals of FPGA-based DNN inference accelerators to achieve both high speed and energy efficiency.

Table 1.2: Main design objectives of FPGA-based DNN inference accelerators for speed (on the left) and energy efficiency (on the right) optimizations.

Increase throughput/reduce latency	**Reduce energy consumption**
Reduce the execution time per MAC by decreasing the critical path of DNN model, to increase the working frequency. *Avoid unnecessary MACs*, to increase parallelism. *Increase PE utilization* by both distributing MACs to all PEs and providing the memory bandwidth to deliver the data to be processed to all PEs, to save execution cycles. *Increase the number of PEs in the system*, in order to perform more MACs simultaneously, to increase parallelism.	*Reduce data movement from off-chip memory to PEs*, maximizing the data reuse. *Reduce the energy cost per MAC* by reducing the precision with which the data to be processed. *Avoid unnecessary MACs*

1.6 Spiking Neural Networks Basic Concepts

Within the field of brain-inspired computing, *spiking computing* is inspired on the fact that communication between biological neurons consists of short electrical pulses (see Figure 1.19) and that information is not encoded just in the amplitude of each pulse, but also on its arrival time.

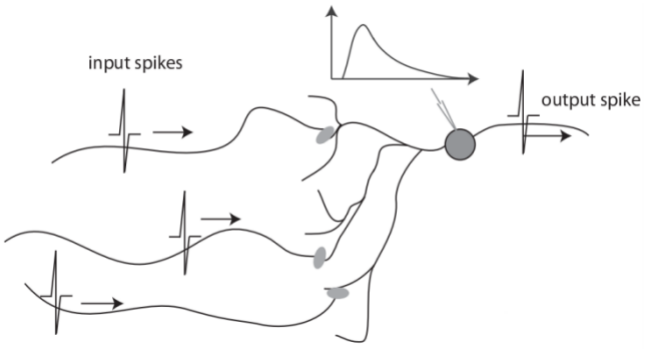

Figure 1.19: Graphical model of a spiking neuron. Source [20]

Spiking Neural Networks (SNNs) [21] are considered as the third generation of ANNs. In recent years, there has been a growing interest in them, because of their extremely high computational energy efficiency when implemented in hardware. While traditional DNNs focus primarily on maximizing model performance and accuracy, SNNs are mainly used for low-power ML applications, due to their spiking nature.

Over time, many different *spiking neuron* models have been proposed with different biological plausibility and computational cost required for their implementation.

The most commonly used models of spiking neuron are the *Hodking-Huxley models*, the *Izhikevich models*, the *Integrate and Fire models*, and the correspondent more realistic *Leaky-Integrate and Fire* (LIF) models. The latter is the most widely adopted structure, being relatively simple and accurate (see Figure 1.20), since it also takes into account the *membrane leakage* present in the biological model.

In general, any spiking neuron model is more biological plausible compared to the traditional non-spiking one, since data transfers among units take place through discrete short pulses called

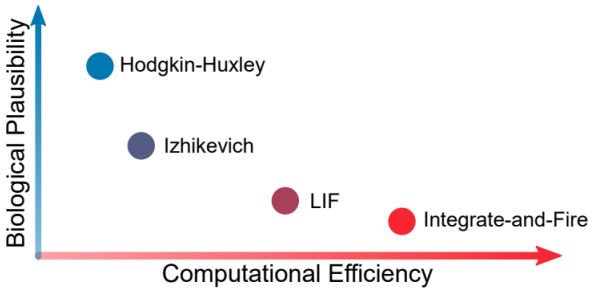

Figure 1.20: Comparison of different spiking neuron models in terms of biological plausibility versus computational efficiency. Source [22]

spikes. In SNNs, the communication between neurons happens through *spike trains*, and thus the information carried by any signal consists in a series of short pulses. An example of a SNN structure and the computational model of a spiking neuron is illustrated in Figure 1.21.

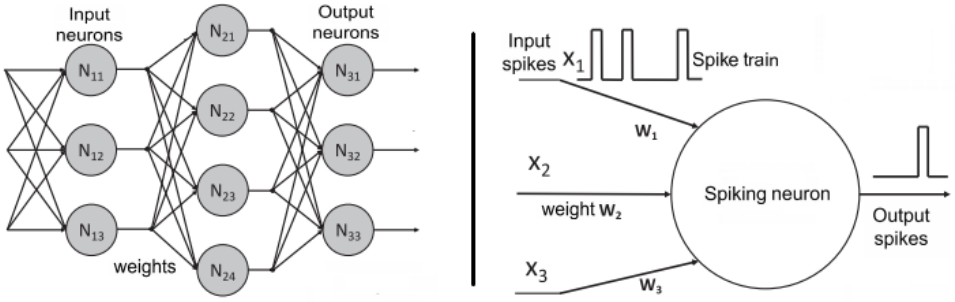

Figure 1.21: Internal structure of a Spiking Neural Network. An example of a Spiking Neural Network (on the left) and computational model of a spiking neuron (on the right). Source [23]

In contrast to traditional DNNs, where each neuron consists in a simple mathematical computational unit that transmits information at each cycle, in SNNs each spiking neuron transmits information in output only when the associated *membrane potential* reaches a specific value, referred to as *threshold* (see Figure 1.22).

More in detail, any spiking neuron receives multiple input signals in the form of short binary pulses, which are first multiplied by the synaptic weights associated to each connection and then integrated on the membrane potential. Thus, only if the result overcomes a specific threshold value the neuron fires, generating a spike in output, and then its membrane potential is reset to a certain value. In addition, the membrane potential decreases over time, due to the *membrane leakage*. After the generation of an output spike, for a determined amount of time called *absolute refractory period* it is not possible for the spiking neuron to produce in another spike signal.

To provide input and generate output spikes correctly to/from SNNs there are several methods based on *frequency* and *temporal-based on coding* to transform continuous numerical values into discrete binary pulses. The most commonly used types of encoding methods are represented in Figure 1.23.

As a result, SNNs are more computational powerful compared to non-spiking ones, due to the possibility to encode temporal and spatial information inside a spiking train. As a result, performing the same task in the former usually requires fewer units than in the latter.

The asynchronous computation in SNNs also leads to a faster inference processing, making

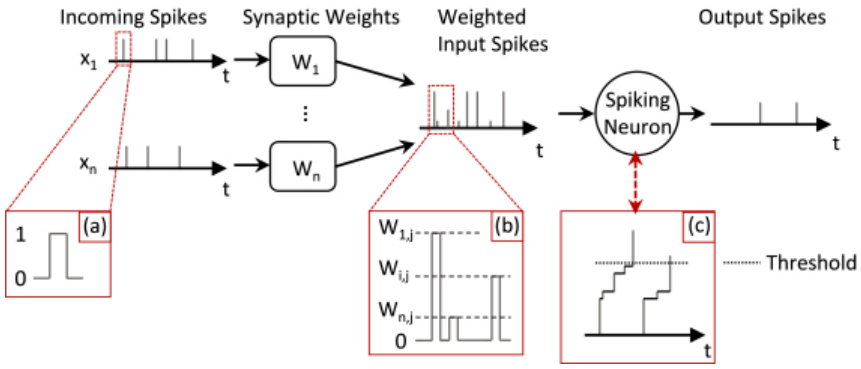

Figure 1.22: Detailed schematic representation of an Integrate and Fire spiking neuron model. Source [24]

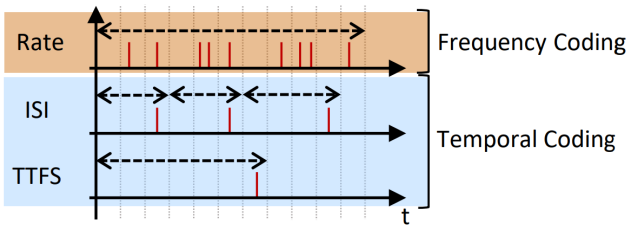

Figure 1.23: Comparison of different encoding techniques for Spiking Neural Networks. Source [25]

them interesting for edge ML applications. While in traditional DNNs all layers of the network must be updated before producing the output, in SNNs a first approximated result is available after receiving the first input spikes.

In addition, also the energy efficiency related to the computational part of SNNs is higher than for the traditional DNNs for two main reasons.

First, in SNNs the execution of an operation happens only when a new spike arrives in the input of a node. This, combined with the fact that each neuron *'fires'* only if its membrane potential crosses the threshold value, makes SNNs sparse, and therefore reduces the number of operations required to perform the same task compared to a traditional DNN model.

Second, the operation types are different in the two cases. While in DNNs the weighted-sum operation are mapped on MACs, in SNNs the encoding of the signal values is binary *spikes* (logic '0' or '1'), which implies that to calculate the output it is required the execution of addition operations between the input weighted spikes and a comparison between the membrane potential and the threshold value. Thus, in SNNs any MAC is simplified to an addition operation, allowing to reduce the computational complexity and energy consumption, since the multiplier is the most expensive computational resource in DNNs. As a result, hardware implementations of SNNs are very attractive especially for low-power embedded applications characterized by a limited amount of computational resources.

Unfortunately, working on binary discrete-valued signals does not allow to achieve the same level of accuracy compared to the continuous-valued case. Moreover, in SNNs it is necessary to store additional parameters (such as membrane potential and threshold values), which results in an increase in memory requirements. As a consequence, SNNs increase the number of required memory accesses compared to standard DNNs, and can be a problem if not handled properly.

Thus, to accelerate SNN processing it is necessary to develop dedicated processing units capable of supporting the implementation of custom hardware logic based on the so-called *neuromorphic computing*. More in detail, the idea is to implement an architecture where the PEs and storage elements are as close together as possible to each other, as for DNN with the memory hierarchy implementation. In such a scenario, deployment of neuromorphic architectures [26] on FPGAs represents an attractive solution in terms of SNN hardware implementation.

Chapter 2

Hardware Design of DNN Processing for Edge Computing

In the following sections, a thorough analysis of the main factors to consider in order to best understand DNN inference processing on embedded systems is carried out.

In order to select the most suitable processing unit for the hardware implementation of DNNs, it is necessary to analyze the DNN model both in terms of internal structure and computation data stream with respect to the amount of computational and memory resources available on the chosen hardware platform, evaluating pros and cons of the different possibilities. The goal is to find the optimal solution capable of both maximizing throughput/latency by mainly exploiting the high parallelism of MAC computations in CONV and FC layers, and improving energy efficiency by reducing the problem due to off-chip memory accesses. To achieve these improvements, it is necessary that the chosen processing unit has a sufficient degree of architectural flexibility to support the application of the optimization techniques introduced for accelerating the DNN inference processing.

2.1 DNN Processing Overview

DNNs are nowadays the most widely used computational models in the most ML applications, due to their ability to achieve an incredible level of accuracy compared to traditional ML algorithms. Over the time, the demand for ever higher performance in terms of accuracy has resulted in the development of ever deeper ANNs as a number of both layers and neurons. It has resulted in increasing demands of computational and storage resources, with direct implications on performance in terms of latency/throughput and energy consumption, and their implementation has become a challenging task, especially for resource-constrained embedded devices.

This conflicts with the current trend of moving the DL algorithm processing to embedded systems, as by running the DNN processing directly at the edge, there is no longer a need to transmit the DNN model and data to be processed to the cloud, saving both latency and the power consumption required for this transfer. But unlike performing on the cloud, where the focus is on the performance, the execution of ML applications at the edge has a whole different set of requirements to meet, focusing on achieving small power consumption and low latency. The limited amount of available hardware resources and the high number of MACs to be performed for DL processing makes hardly possible to directly implement DNN models on embedded devices respecting these constraints, maintaining also a high level of accuracy.

Any DL algorithm consists of two steps, which are the training, to determine the network parameter values, and the inference, to evaluate the trained model to produce the results for the

task in exam , with the former much more expensive in terms of both computational and memory requirements than the latter.

To have an idea, the training demands the execution of approximately the double one of *multiply-and-accumulate* (MAC) operation regarding the inference, which is an operation consisting of a multiplication and an accumulation sum. Moreover, while the training needs an extremely high precision to determine the value of the network parameters, as it directly affects the model accuracy, the inference requires less precision, since it is less sensitive to a lowering of precision in data representation. Thus, as DNN models are enough robust in the production of the correct result in inference to a reduction of the precision on weights and activations to be processed, it can be performed at lower precision than in training.

In addition, performing a higher number of computations also results in a higher number of memory accesses to read/write the data to be processed. More in detail, the execution of each MAC operation requires up to 4 memory accesses, so the interfacing of the processing elements with the storage device, in addition to being the most significant contribution in terms of power consumption, can be a limiting factor for the execution of DL algorithms due to memory bandwidth limitations.

For these reasons, most of edge hardware accelerators only perform the inference, while the training, remains devolved to a cloud-based execution.

Thus, as DNN models increase in complexity and consequently integration on embedded systems becomes increasingly difficult for low latency and low energy consumption ML applications, possible solutions must necessarily include at the architectural level the development of highly parallel paradigms with efficient data flows to improve speed and energy efficiency, and the development of optimization methods to reduce computational and memory requirements.

At the level of hardware platform for edge DNN inference processing, the choice consists of *software-based* and *hardware-based* solutions, referred to as *hardware DNN inference accelerators*. While the former offer the flexibility to implement multiple types of ML task by sequentially executing the corresponding DL algorithm instructions, the latter exploit the inherent parallelism of any DNN model improve the performance. For these reasons, the hardware platforms must be flexible enough to take advantages of these features by supporting the application of a variety of optimization methods. As a result, the degree of re-configurability and architectural flexibility of the processing unit on which to implement the DL algorithm is an essential feature in order to accelerate the DNN inference processing. Consequently, in order to find the most convenient solution, it is important to analyze how the flow of operations execution required for DNN inference processing differs depending on the type of hardware platform selected.

More in detail, since the depth of DNN models in terms of number of layers may reach a considerable size, it results in the requirement of a large number of MACs and memory accesses for DNN processing. Although DNNs are composed of different types of layers, operations and parameters are mainly in FC and CONV layers (see Figure 1.15), and thus the optimization of computation in these two kinds of layers is the main goal in any hardware DNN inference accelerator designs.

2.1.1 Analysis of the Multiply-And-Accumulate Operation

In DL processing, most of the internal computations in CONV and FC layers consist of performing multiplications between activations and weights, and then accumulating the obtained results. A possible solution to implement in hardware the weighted sum operation is the MAC operator.

Although a MAC operation is not too complex, the amount of times that must be performed during DNN processing and the number of memory accesses to take data to be processed is extremely high. More in detail, in the worst case the execution of a MAC operation in FC and CONV layers requires four memory accesses, which are three memory reads and one memory

write, as shown in Figure 2.1.

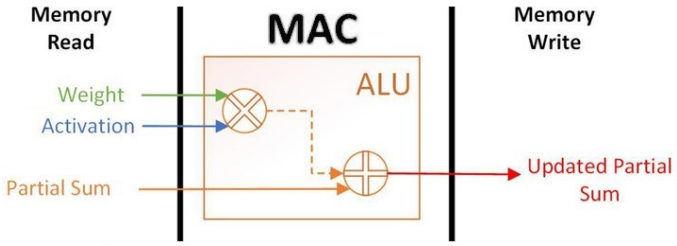

Figure 2.1: Memory accesses for multiply-and-accumulate (MAC) operation execution. Source [27]

With the goal of accelerating the DNN processing, it is possible to parallelize the independent MAC operations with a DL algorithm hardware implementation. To do this optimally, processing units with highly parallel computational paradigms have been developed over time, capable of handling data flows[1] efficiently in terms of energy consumption. More in detail, the internal architecture of any processing unit is structured as an array of *processing elements* (PEs), where each one corresponds to a computational unit capable of performing a single MAC operation.

Because of the high number of MACs for DNN inference iteration, the size of the on-chip memory of a hardware accelerator is usually not sufficient to store all the parameters to be processed. Consequently, interfacing with a larger off-chip memory becomes necessary. Usually the on-chip memory is a small and energy efficient SRAM, while the off-chip memory is a large DRAM. More in detail, in case the memory access takes place through the latter, which being larger and more distant than the former to the PE array determines both a higher energy cost and a longer latency, causing a further reduction in the performance and energy efficiency of the system. As an example, the results reported in [28] show that the amount of energy needed to perform an access to a data item contained in standard off-chip DRAM memory is about three orders of magnitude larger than that required to perform a MAC operation in *single-precision floating point data format* (fp32), and almost two orders of magnitude larger than that required to perform an on-chip SRAM memory access.

Thus, for DL algorithm processing, the main limitation is not in the computational part, but in the interfacing between PEs and memory. Thus, in order to enhance DNN inference processing it is mandatory to optimize memory accesses, since performing off-chip memory accesses compared to on-chip memory accesses has a significant impact on the DNN model performance both in terms of latency and energy consumption.

To mitigate this problem, it is possible to allocate a certain amount of local memory on-chip, usually organized in hierarchical levels, which is smaller, but faster and more energy efficient than the global one. The idea is to leverage the local memory for maximizing the *data reuse*, so that data read from DRAM and moved to SRAM memory hierarchy are used multiple times, and not just to perform a single MAC operation. Figure 2.2 compares how the memory operations required for a MAC execution are done in the case with and without local memory hierarchy, respectively.

Table 2.1 shows a performance comparison of two standard types of on-chip SRAM to off-chip DRAM used in DNN inference hardware accelerators.

[1]The spatial and temporal mapping of MAC operations required for DNN processing is called *data flow*.

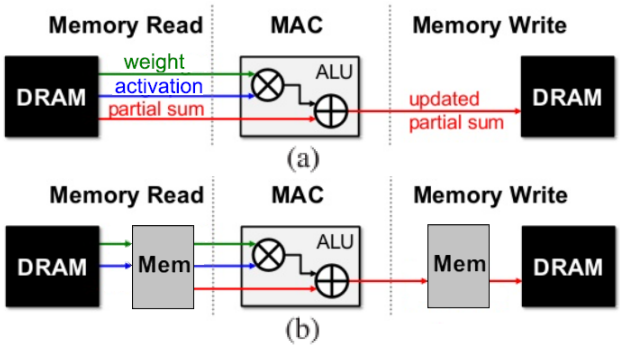

Figure 2.2: Read/write operations in memory required for MAC operation execution. (a) Standard MAC operation (b) MAC operation with local memory hierarchy. Source [29]

Table 2.1: Comparison of on-chip SRAM and off-chip DDR SDRAM. Source [30]

	On-chip SRAM	*Off-chip DDR DRAM*
Energy/bit (pJ/bit)	0.0005	0.05
Read time (ns)	0.3 to 1	10
Write time (ns)	0.3 to 1	10

2.1.2 Approximate Computation for DL Algorithms Overview

Regarding the design of hardware DNN inference accelerators for edge computing, the development of approximation methods capable of making the hardware implementation of DNNs more feasible by reducing the demand in terms of model complexity is of increasing interest.

More in detail, solutions capable of reducing the DNN model complexity both in terms of precision of weights and activation and in the number of operations and model size with the aim of improving the performance and energy efficiency of the hardware accelerator at the expense of a certain loss of accuracy are fundamental to implement computational-expensive and memory-intensive DL algorithms on embedded devices.

While the reduction of the precision of the operands/ operators determines a worsening of the final accuracy compared to the full precision DNN model, it also allows to obtain an improvement in terms of throughput, memory bandwidth and size and energy efficiency of the system. More in detail, it is mainly due to for the following reasons:

1. A lower demand in terms of memory demand, which therefore results in a reduction in the physical size of the memory, and which in turn results in an improvement in terms of power consumption and latency per memory access (smaller and closer memory to PEs are faster and more energy efficiently than larger and more distant memory).

2. A lower computational cost per MAC execution, resulting in a reduction in both the area required to implement a MAC operator and the energy consumption needed to execute a MAC operation, resulting in an increase in the execution speed and energy efficiency.

3. A reduced bit width for the representation of the weight and activations to be moved from memory to PEs results in a reduced power consumption per data transfer and higher speed, and thus in a greater amount of information transferred with the same available memory bandwidth.

Similarly, reducing the number of operations and the model size results in a less computational and memory resource use per DNN inference iteration, which translates into improvements

in both of throughput/latency and energy consumption. Moreover, through the application of approximation techniques that avoid to perform operations that do not contribute significantly to the model accuracy by eliminating the corresponding operands/operator, it is possible to avoid the memory accesses to the parameters to be processed, thus also reducing the memory bandwidth and the storage resource requirements.

More in general, for DL algorithm inference processing at the edge, approximation methods can be applied at different levels of abstraction, which specifically are *circuit-level approximation*, *architectural-level approximation*, and *system-level approximation*, as shown in Figure 2.3.

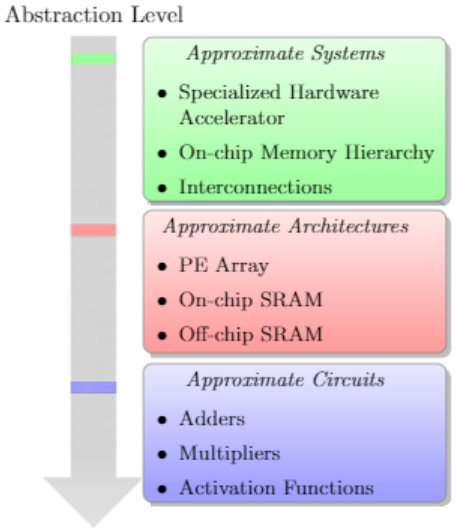

Figure 2.3: Classification of approximation methods at different abstraction levels for Deep Neural Network hardware implementation.

At the *circuit-level*, since during the DNN inference processing the MAC results to be the most performed one, its approximation allows both to reduce the energy consumption and increase the performance relative to the computational part, at the expense of a certain loss of accuracy on the produced result.

At the *architectural-level*, approximating the PE array through the use of structures composed of an higher number of MACs allows both to reduce increase the throughput and to reduce the power consumption, at the cost of introducing some loss on the accuracy of the final result.

At the *system-level*, a wide range of hardware accelerators for the DNN inference processing based on different hardware platforms have been developed able to improve performance in terms of both energy efficiency and speed. In addition, it is particularly important to implement an on-chip memory hierarchy to reduce the number of accesses to off-chip memory, maximizing the reuse of data on-chip.

2.2 Hardware for DNN Processing: Energy Efficient Architectures

Over time, several efficient specialized hardware accelerators for DNN model have been proposed, consisting of highly parallel computational computing paradigms capable of handling the stream of computation inside DNN models with high energy efficiency, maximizing the reuse of the different type data for MACs in CONV and FC layers [25].

The possible types of hardware platforms able to accelerate DNN inference are shown in Figure 2.4.

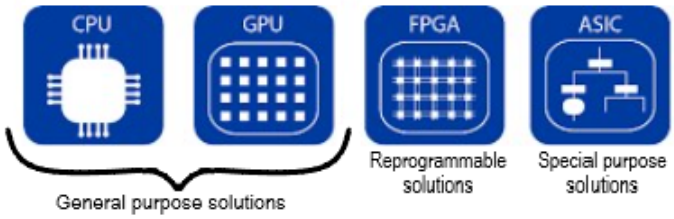

Figure 2.4: Overview of different hardware platforms for DNN processing at the edge. Source [31]

The optimal choice depends on the DL algorithm phase to accelerate and the corresponding ML application constraints. With the goal of designing specialized hardware accelerators that can achieve both high speed and energy efficiency, it is important to understand how the internal execution differs depending on the selected processing unit.

In more detail, to exploit the inherent concurrency in the execution exhibited by any DNN model paralleling the MAC operations as much as possible, two types computing paradigms are commonly used, referred to as *spatial* and *temporal architectures*, respectively (see Figure 2.5).

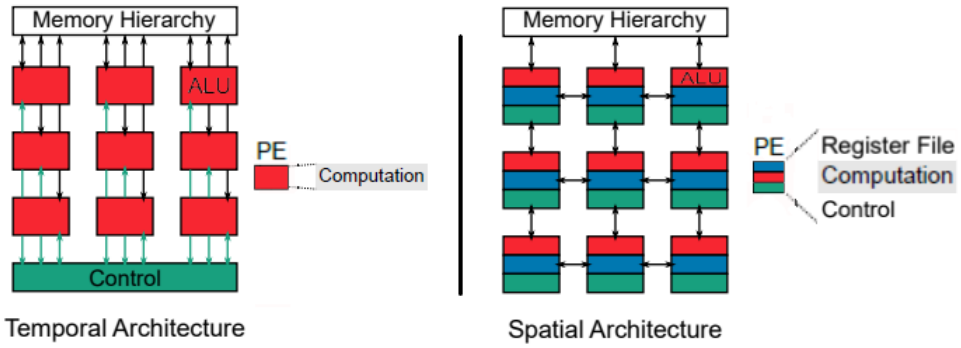

Figure 2.5: Parallel computing paradigms consisting of an array of processing elements (PEs): temporal hardware architecture (SIMD/SIMT) (on the left) versus spatial hardware architecture (data flow processing) (on the right). Source [22]

Although the architectural structure of these two solutions is similar, a more detailed analysis reveals some substantial differences, which are:

1. *Temporal architectures*, where the PE array consists of a structure of *arithmetic logic units* (ALUs) capable of performing multiple MACs concurrently, managed with a global control logic. The direct communication between different PEs is not possible, and the memory accesses to the data needed to perform DNN processing is only through the external memory hierarchy, limiting the performance and energy efficiency of the system. Hardware DNN inference accelerators based on CPUs and GPUs belong to the category of temporal architectures. While they are able to execute a wide range of ML tasks, thus maintaining a very high level of flexibility, on such hardware platforms optimizing the execution of a specific DL algorithm is not so easy, due to their fixed architecture. As a result, it is the DNN model to execute that adapts to the processing unit, and not the other way around. In this case the idea to exploit the intrinsic concurrency in execution of DNN models is to parallelize the MACs by adopting efficient computing paradigms, such as *single instruction multiple data*

(SIMD) for CPUs (i.e. multiple PEs perform the same instruction on multiple data simultaneously) and *single instruction multiple threads* (SIMT) for GPUs (i.e multiple PEs perform the same instruction on multiple threads simultaneously on different data).

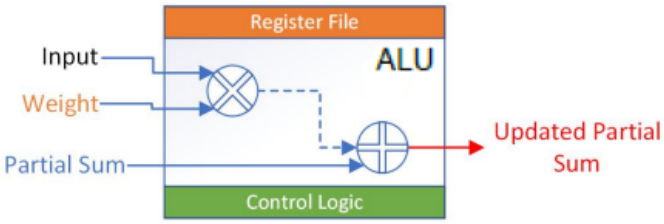

Figure 2.6: Internal structure of a PE unit in spatial architectures capable of performing an individual MAC operation, consisting in a multiplier, an adder, a register file and some local control logic. Source [27]

2. *Spatial architectures*, where each PE consists of an ALU with associated its own local control logic and its local memory, referred to as *Register File* (RF) (see Figure 2.6). As a result, in this case it is possible to directly move data between different PEs (through the so-called *inter-PE communication*) in an optimal way and to develop energy efficient data flows to reduce the accesses to the external memory. This is achieved by increasing the reuse of data distributed in the low levels of memory hierarchy as much as possible. It allows to improve both the performance and the energy efficiency of the system. Hardware DNN inference accelerators based on FPGAs and ASICs belong to the category of spatial architectures. While the architectural flexibility of FPGAs due to their re-programmability and re-configurability allows for a custom configuration that can be changed over time, ASIC-based solutions, must be designed once for the specific application.The processing units based on spatial architecture can adapt better to follow the parallel execution model of DL algorithms, supporting also a wide range of optimization techniques.

Table 2.2 summarizes the main differences between spatial and temporal hardware architectures.

Table 2.2: Main differences between temporal and spatial architectures. The two architectures consist of an array of PEs that perform multiple MAC operations in parallel.

Temporal architecture	**Spatial architecture**
PEs include only the computational part (ALU)	PEs include a computational part (ALU), local memory (RF) and local control logic
Only global control logic	Both global/local control logic
Only data movement from PEs to memory hierarchy (and vice-versa)	Data movement between different PEs and from/to memory hierarchy

Thus, in processing units based on spatial architecture where the control logic is local, the solution adopted to mitigate the problem of energy consumption due to data movements from memory to PEs consists in reducing the number of accesses to the off-chip memory, having it both a higher latency and energy cost than the on-chip one. Thus, the idea is to develop hardware DNN inference accelerators with an internal memory divided in multiple levels of hierarchy.Consequently, spatial architectures are a more cost-effective choice than temporal architectures, because by distributing part of the internal memory directly within each PE, they reduce the impact of memory accesses in terms of both latency and power consumption, and allow maximizing the reuse of the data.

A typical structure of a hardware DNN inference accelerator based on the spatial architecture adopting a on-chip memory hierarchy is shown in Figure 2.7.

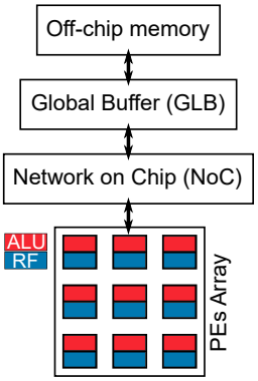

Figure 2.7: Internal structure of a hardware accelerator based on spatial architecture for DNN inference processing, consisting of an array of PEs. Source [32]

More in detail, the on-chip memory hierarchy is composed of a *Register File* (RF) internal to each PE, containing the data to move among different PEs or MAC results, and *Global Buffer* (GB) directly connected to the off-chip memory, with a size large enough to store all the network parameters needed to feed all PEs. The various PEs are connected to each other and to the GB by a configurable *network-on-chip* (NoC).

Thus, during the DNN inference processing the goal is to maximize the reuse of data stored in the lower levels of the on-chip memory hierarchy, while avoiding access to off-chip memory unless strictly necessary (see Figure 2.8, on the top). Thus, this strategy allows for increased throughput and reduced energy consumption per DNN inference iteration, since the various layers of the local memory hierarchy have different latency and energy cost from each other, as shown in Figure 2.8 (on the bottom).

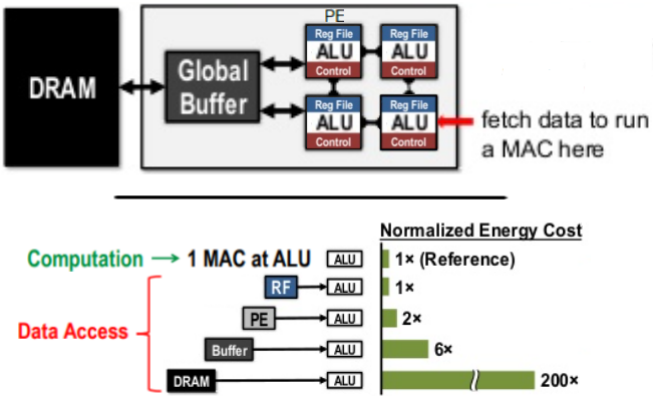

Figure 2.8: Memory hierarchy in spatial architectures (on the top) and normalized energy cost for MAC and memory accesses to data stored in different levels of the memory hierarchy (on the bottom). Source [28, 33]

More in detail, in DNNs there are different types of data that can be reused, depending on the considered type of layer. In CONV layers it is possible to identify the *weight reuse* (i.e. the weights of the kernel sliding on an input feature map are reused a number of time equal to the

corresponding output future map dimension), *input reuse* (i.e. each input feature map is reused a number of time equal to the number of filter channels to compute the corresponding output future map), and *convolutional reuse* (i.e. the input feature map values involved multiple times in the sliding operation of convolutional kernel are reused to compute multiple output feature map values). In FC layer it is possible to identify the *input reuse*, since each neuron uses all the inputs to compute its output.

As a result, the data reuse is higher in CONV layer than in FC layers. It is important, since CONV layers is where most of the MACs are performed. Although in FC layers the reuse of the data is not so high, this is not a problem, because with the increasing of the depth of the DNN model is the number of CONV layers to increase.

Regarding the design of any hardware DNN inference accelerator, another important aspect to consider is that the amount of PEs is limited. Consequently, only a number of MACs equal to the amount of available PEs can be executed in parallel, which is usually less than the total number of MACs required to perform a DNN inference iteration. The mapping among data to be processed and operation is defined by configuring the NoC in the proper way. It allows to to transfer the activations, weights, and partial sums from the memory to the different PEs, with the goal of maximizing the reuse of the data at the lower levels of the memory hierarchy. Moreover, it allows also to accelerate the DNN inference processing reducing the memory bandwidth requirements, since the data read from RFs within the PEs limit the number of access to off-chip memory. Specifically, if the memory is not able to feed all the available PEs, the performance of the hardware accelerator decreases.

Depending on which type of data are stored in the RFs, and so on the type of data reuse exploited, several types of data flows can be identified (see Figure 2.9).

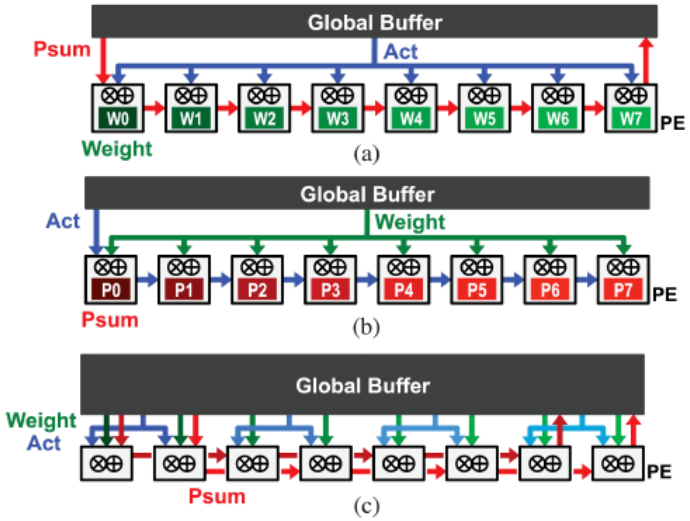

Figure 2.9: Possible data flows in spatial architecture for Deep Neural Networks. (a) Weight stationary (b) Output stationary (c) No local reuse. The acronyms Weight, Act, and Psum stand for weights, activations and partial sums to be read from memory to perform MAC operations or to be written in memory as a result of MAC operations, respectively. Source [33]

More in detail, it is possible to classify the different data flows for hardware DNN inference accelerators based on spatial architectures as:

– *Weight stationary* (WS), where the goal is to reduce the energy consumption maximizing the reuse of the weights by storing them in the local RFs of each PE, while activations and

partial sums are both read from/written to the GB.

- *Output stationary* (OS), where the purpose is to reduce the energy consumption, and consists in storing the partial sums in the local RF of each PE, without reading/writing them from/into the GB, which is instead needed for activations and weights.
- *No local reuse* (NLR), where the goal is to reduce the area, and it is done by removing all RFs of each PE in order to maximize the GB size, at the cost of increased energy consumption. In this case, therefore, data reuse is not exploited at all, and the entire available area is allocated to GB, where weights, activations and partial sums are all stored.
- *Row stationary* (RS), where the goal is to reduce the energy consumption by maximizing the reuse of all types of data by dividing the convolution operation among activations and weights by rows to perform it on the same PE (see Figure 2.10).

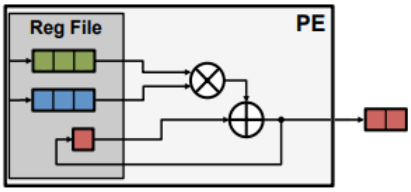

Figure 2.10: Convolutional operation division by row in the same PE in Row stationary data flow. Source [33]

Since the energy costs of the different level of on-chip memory hierarchy (see Figure 2.8) is different, the main goal of these optimized data flows is to maximize the reuse of data in low levels of on-chip SRAM memory hierarchy, to minimize the number of accesses to the off-chip DRAM. As an example, a comparison of the different data flows in terms of energy consumption with respect to the MAC energy cost in AlexNet is shown in Figure 2.11. It results that the RS data flow is the best possible one in terms of energy efficiency improvements, allowing optimization of the memory access cost for all the types of data.

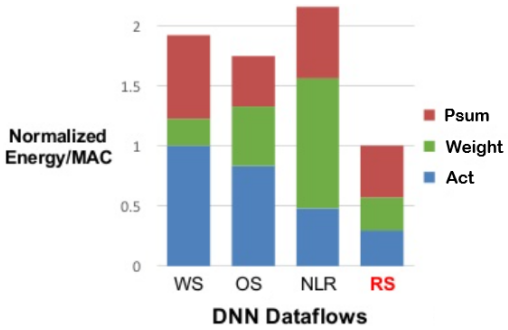

Figure 2.11: Comparison of different data flows in terms of energy efficiency in AlexNet. Source [33]

2.2.1 Strategies for accelerating spatial architecture-based accelerators

Over time, by taking advantage of the architectural flexibility of hardware accelerators based on spatial architectures, more complex optimization have been developed to improve the speed and energy efficiency of DNN inference processing, which are:

– *Sparsity*

The computations in CONV and FC layers performed during DNN processing typically involve a large number of multiplications in which at least one of the two operands is null. In such cases, since the result is known in advance, it is not necessary to perform the corresponding MAC operation, resulting in a significant reduction of the number of MACs per DNN inference iteration. Following this idea, it is possible to develop techniques with the aim to increase the level of *sparsity* of a DNN model, such as the so-called *network pruning* and *activation function statistics*. In addition, it is possible to modify the structure of each PE to not read the weights/activations from memory and to execute the MAC operation when one of the two operands to be processed has zero value (see Figure 2.12).

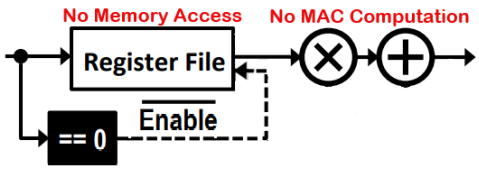

Figure 2.12: PE internal structure changes to skip the memory read operation and MAC calculation when at least one operand is zero. Source [34]

This strategy allows not only to skip the execution of a large number of MACs, and so to reduce the memory bandwidth requirements, but also to compress the data to be stored in memory through the so-called *sparsity compression methods*. Hardware accelerators that exploit the sparsity of the DNN model to optimize inference processing in terms of both speed and energy efficiency require a high degree of flexibility to adapt the internal architecture to a sparse computation. It may require the implementation of custom hardware capable of distinguish among the the necessary and unnecessary MACs to be performed, through the so-called *zero-skipping* methods. For this reason, FPGAs is the optimal solution for hardware accelerators that want to take advantage of sparsity, due to the possibility of designing custom hardware for optimizing sparse computation.

– *Variable bit widths*

The more the complexity of DNN models increases, the more the number of operations to be processed increases, and consequently the amount of memory required to store the data to be processed during the inference becomes high. As a result, not only the computational cost increases, but also becomes difficult for the on-chip memory to be sufficient for storing all these data, resulting in an increase in the number of accesses to the high latency and high energy consumption off-chip memory. A possible method to reduce the memory occupancy and the computational cost is to change the data type and reduce the number of bits on which the data is represented. While floating point representation allows for higher precision, it also results in a high hardware complexity for MAC implementation and higher power consumption for operation execution. Through the *data quantization* it is possible to move from floating point to lower bit width fixed-point data format, at the expense of a certain loss of accuracy. This allows to obtain improvements in terms of both computation complexity and memory requirements, allowing to improve speed and energy efficiency of the hardware accelerator, at the cost of some loss in model accuracy. In addition, depending on the network parameter type and position within a DNN model, each layer has a different impact on the system accuracy, and thus it is useful to have the ability to not only choose a different number of bits for different network parameter types belonging to the same layer, but also for the different layers in the same DNN. It allows to have smaller DNN models with the

same level of accuracy than with fixed bit width for all the DNN model, by reducing also the memory bandwidth requirements. Thus, to take advantage of the reduction in bit length for data representation in terms of both memory requirements and power consumption, *hardware DNN inference accelerators with variable bit widths* have been developed over time. The configurable architecture of FPGAs allows to design logic units optimized for the computation on different number of bits, resulting to be the most promising processing unit for DNN inference accelerators with variable bit widths.

2.3 Hardware for DNN Processing on Embedded Devices

Over time, as the amount of the data to be processed and the size of the DNNs have become larger and larger, to improve performance of DL processing it has been required the development of specialized hardware accelerators for implementing DNN models.

As a result, the choice of the processing unit on which to base the development of these hardware DNN inference accelerators is of paramount importance. More in detail, it depends where the ML application is performed and on the required performance and the consumption requirements of the case at a hand.

While the training is usually performed on GPUs on the cloud optimized for high precision floating point computations, the inference requires much less computations. Thus, it can be performed also on a different type of processing unit at the edge, such as *FPGAs* and *ASICs* for ML applications, called *TPUs*, depending on the requirements to be undergone (see Figure 2.13). To have an idea regarding the performance, while edge hardware accelerators for DNN models typically range from 100 GOP/s to 10 TOP/s, hardware acceleration on the cloud may exceed 100 TOP/s.

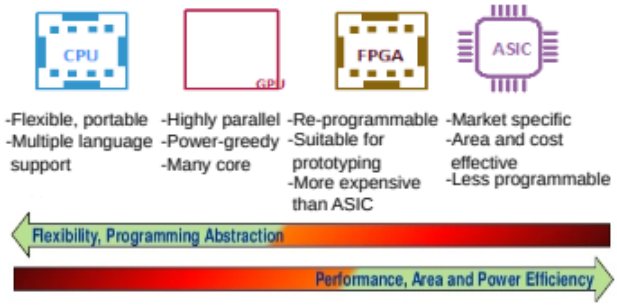

Figure 2.13: Comparison of different hardware platforms in terms of flexibility and programming difficulties versus performance, area and power efficiency. Source [35]

Unlike CPUs and GPUs which are based on a fixed architecture, FPGAs and ASICs allow to realize flexible architectures optimized for the application at hand thanks to the possibility to implement customized logical units and to support the application of a wide range of optimization algorithms. More in detail, while ASICs are completely non-reconfigurable, FPGAs contain reconfigurable custom hardware logical units that make possible to obtain high performance allowing a run-time development of solutions better and better optimized for the ML application under examination. Furthermore, despite a much low working frequency compared to alternatives, both of these solutions are able to take advantage of the parallelism of calculations in DNN inference processing through pipelined execution in hardware.

To have a general idea, CPUs are capable of performing a number of operations per second of about 1 GFLOP/s to 10 GFLOP/s, with an energy efficiency usually lower than 1 GOP/J, while GPUs allow the execution of a number of operations per second up to 1 TOP/s, but with much lower energy efficiency. FPGAs can achieve at least 10 GOP/s to 100 GOP/s, with much higher

energy efficiency than all GPPs [12]. For ASICs, in terms of speed, these solutions are somewhere between those based on GPUs and FPGAs, but in this case the achievable energy efficiency is the best possible one. However, the behavior of these processing units in terms of flexibility is exactly the opposite of that in terms of performance, as shown in Figure 2.14.

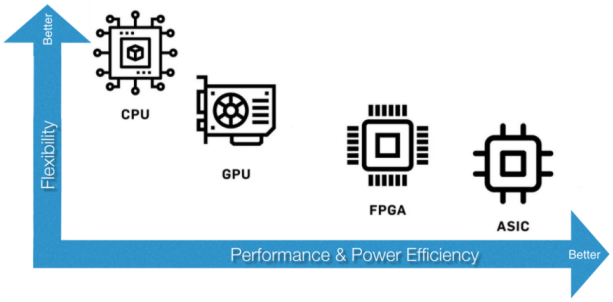

Figure 2.14: Comparison of CPUs, GPUs, FPGAs and ASICs in terms of flexibility versus performance and power efficiency. Source [36]

The idea behind the accelerated execution of the DL inference in hardware is to exploit the inherent parallelism exhibited by DNN models. Any DL algorithm presents different types of parallelism that must be considered to determine the most suitable hardware platform able to exploit them during DNN inference processing.

More in detail, it is possible to parallelize the execution of DL algorithms both at *layer level* (which means that in a multi-layer ANN it is possible to process different data-dependent layers in parallel exploiting a *pipeline* execution), at *level of neurons* (which means that since neurons in the same layer are independent from each other, they can work in parallel) to increase the level of concurrency in the execution of the MACs within CONV and FC layers, and at the *input level* (which means more input data can be processed simultaneously).

More in detail, the possible processing units for accelerate the DNN inference processing are:

1. *CPUs - Central Processing Units*, which are GPPs composed of several cores capable of processing DNN inference as a sequential flow of instructions. Their high degree of flexibility to execute instructions related to different types of tasks in turn results in a decrease in performance for the task at hand. Despite a high operating frequency, the lack of parallelism exploitation determines that the maximum processing speed achievable by CPUs is usually not sufficient to perform the amount of computations required per DNN inference iteration for low-latency applications. In addition, power consumption is also an issue for these solutions, making them the least used for edge DNN processing.

2. *GPUs - Graphic Processing Units*, which are GPPs specifically optimized for high precision floating point computation, composed of a much higher number of cores than CPUs, but with a slower working frequency. The idea in this case is to take full advantage of the level of execution concurrency exhibited by DNN models through a processing unit with a high level of parallelism and a high memory bandwidth, thus able to perform the MACs required per DNN inference iteration very fast. As for CPUs, also in this case the execution is at instruction level in a sequential way. Compared to CPUs, GPUs accelerate the execution of the MACs, maximizing the throughput, at the expense of a huge energy consumption. As a result, while GPUs are the optimal choice for DNN processing on the cloud due to their high computational power, they are not used in edge computing due to the limited amount of energy budget.

3. *FPGAs – Field Programmable Gate Arrays*, which are integrated circuits that can be reprogrammed several times, providing the ability to configure the logical units required to implement DNN models in a custom way, optimizing the internal architecture for the specific task at hand. Moreover, the FPGA implementation of DL algorithms offers a viable solution to accelerate the DNN inference, being that the structure of such type of processing units can be adapted to exploit the inherent DNN model parallelism optimally. The limited amount of hardware resources, the low bandwidth of on-chip memory, and the slow operating frequency makes difficult the implementation of complex DNN models on FPGAs. More in detail, although FPGAs do not provide the same level of speed as GPUs if the DL algorithm implementation is direct and if computation required per DNN inference iteration is performed in high floating point precision, their high architectural flexibility allows to support a wide range of optimization techniques able to reduce DNN model complexity and to improve the speed and energy efficiency of the implemented DNN. Thus, while maintaining an acceptable design cost, FPGAs allow to achieve both an adequate level of performance for most ML applications, and a much higher energy efficiency than in the case of GPUs, provided that some decrease in the level of flexibility with respect to the range of executable tasks for the same architectural configuration is accepted. As a result, it makes the implementation of compressed DNN models on FPGAs the most promising choice in edge computing for DNN inference acceleration.

4. *ASICs - Application Specific Integrated Circuits*, which are integrated circuits in which the logical hardware units are programmed to optimize the execution of a specific application, and thus able to provide the optimal in terms of both speed and energy efficiency. The so-called *Tensor Processing Units* (TPUs) are specific types of ASICs designed to perform the DNN processing. TPU DL algorithm implementation usually adopt a lower-precision data type representation than the standard single-precision floating-point representation used in GPPs. As they are not re-programmable at all, the architecture of ASICs is designed to perform only a specific task in the most efficient way, and therefore although they are the most performing hardware platforms for DNN inference processing, they represent the least flexible option in terms of executable tasks. In addition their high design cost limits their range of application in edge computing.

As an example, Table 2.3 shows a comparison of the performance in terms of inference time and power consumption of the different processing units for the case of DNN processing acceleration in AlexNet. The results reported in [37] show that the execution of the DNN inference with respect to a not too complex DL algorithm optimized for implementation on FPGA is about ten times more efficient in terms of power consumption than that performed on GPU, while at the performance level it does not differ much from that performed on TPU.

Table 2.3: Performance comparison of available hardware platforms to accelerate AlexNet inference processing. Source [38]

	CPUs	*GPUs*	*FPGAs*	*TPUs*
Inference time (ms)	1630	3.79	15	4.31
Power consumption (W)	104	206	27	12.5

Since any processing unit has its benefits and limitations, in order to make a fair comparison regarding hardware platforms capable of accelerating DNN inference processing, it is necessary to consider the trade-off between *flexibility*, *performance* and *energy efficiency* of each one.

For embedded implementation of DNN models, while the GPPs have an high degree of flexibility in terms of executable tasks, the fact that they are based on a fixed architecture non configurable determines a decrease in terms of parallelism exploitation of DNN models.More in detail, while CPUs do not allow to reach an adequate level of performance in terms of speed and energy efficiency, GPUs, despite being computationally extremely computational powerful thanks to a massive concurrent execution that allows to obtain very high speed for DNN inference processing, are very inefficient in terms of power consumption and space occupation.ASICs, despite providing the possibility to achieve very high performance in terms of both speed and energy efficiency, are extremely expensive to design, and do not allow to be reprogrammed. Consequently, TPUs do not support the application of run-time optimization methods.

As a result, FPGAs are the best compromise in order to accelerate DNN inference processing at the edge, supporting the application of a wide range of optimization methods able to reduce DNN model complexity and increase speed and energy efficiency. It is of paramount importance in embedded ML applications, where the hardware resource and power consumption is the main issue. Through FPGA-based solutions it is possible to exploit all types of parallelism exhibited by DNN models optimally, while implementing hardware logic units specifically optimized for the considered ML application.

Figure 2.15 gives an example to illustrate the difference between how parallel execution is handled on different processing units for DNN processing. Specifically, it refers to the throughput achievable on 10 clock cycles of a 5-layers CNN [39], which are the CONV layer, ReLU layer, NORM layer, POOL layer and FC layer (see Section 1.4) for the possible parallelization schemes adopted by the different hardware platforms for the DNN inference processing. Furthermore, it is assumed that each layer has an execution time equal to 1 clock cycle, and that each block represents a single set of input data to be processed at a time.

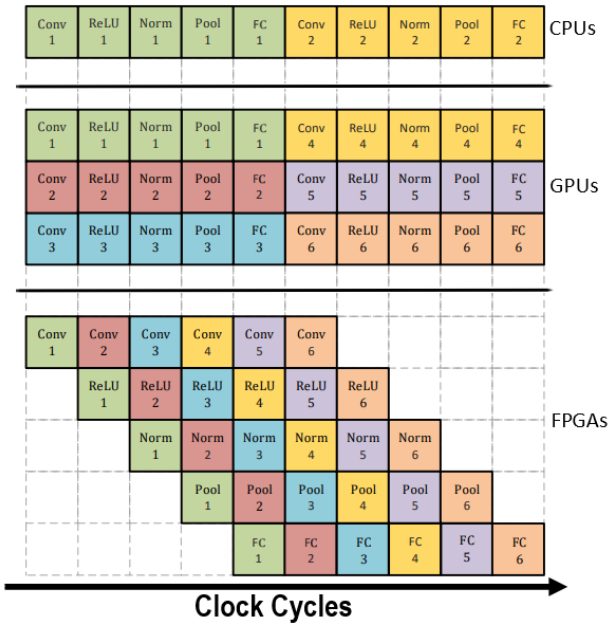

Figure 2.15: Comparison of a 5-layer CNN processing on CPUs (on the top), GPUs (in the middle) and FPGAs (on the bottom) as a function of the adopted parallelization scheme by the different processing units.

- For CPUs, which adopt the SIMD parallel computing paradigm, execution of CNN instruction is performed as a single processing thread at a time on multiple data, and thus two sets of input data are processed over 10 clock cycles, resulting in a throughput of 2 per processing core.

- GPUs exploit the parallelism inherent in CNNs through the SIMT parallel computing paradigm. Assuming that 3 different processing threads of the CNN instruction can be processed in parallel, a total of 6 set of input data are processed in 10 clock cycles, achieving a throughput equal to 6.

- FPGAs parallelize the execution of the CNN model through a pipelined parallel computing paradigm, processing the same amount of input data in 10 clock cycles as GPUs, thus allowing for the same value of throughput. However, if the CNN execution is continued and then additional input data sets need to be processed, GPUs would still require 5 clock cycles to produce 3 new outputs, while FPGAs would provide a new result in output with each clock cycle, resulting in a substantial increase in throughput.

Despite the massive exploitation of the internal parallelism of DNN models allowing high performance and low power consumption, FPGA-based DNN inference accelerators have a high cost in terms of reconfiguration and are complex in terms of programming.

Table 2.4 shows a comparison of the main differences in performance and hardware features of the processing units used to accelerate DNN inference processing.

Table 2.4: Performance comparison of CPUs, GPUs, FPGAs and ASICs for accelerating DNN inference processing at the edge. FPGAs represent the optimal compromise between low latency/low power consumption, small hardware size, desing cost, high level of re-programmability and architectural flexibility, resulting to be the most promising solution to perform DNN inference processing on embedded systems.

	CPUs	*GPUs*	*FPGAs*	*TPUs*
Throughput	Lowest	Very High	High	Highest
Latency	Highest	Low	Very Low	Lowest
Parallelism	Low	Highest	High	Very High
Energy Efficiency	Low	Lowest	High	Highest
Design Cost	Lowest	Low	Medium	Highest
Power Consumption	High	Highest	Low	Lowest
Device Size	Small	Large	Very Small	Smallest
Architecture	Fixed	Fixed	Re-Configurable	Configurable
Adaptability	Highest	Very High	Low	None
Development	Easiest	Easy	Hard	Medium
Inference Execution at the Edge	Low Performance	High Power Consumption	Best Compromise	High Design Cost/ No Flexibility

In conclusion, given that CPUs represent the worst possible solution at the level of edge DNN inference acceleration due to their low level of concurrent execution and low energy efficiency, even if FPGAs do not allow to reach a level of speed comparable to that of ASICs and GPUs, this type of solution allows to achieve much higher energy efficiency (compared to GPUs), and better flexibility at a lower design cost (compared to ASICs). Thus, although for direct implementations of DNN models FPGAs are not the best solution, the high degree of architectural flexibility and re-configurability supports the run-time application of some *hardware/software-level optimization algorithms* and *architecture-level design improvements*, which enable substantial performance and energy efficiency enhancements.

Moreover, the working frequency of FPGAs is lower than that of alternative solutions, resulting in both better energy efficiency, but in a worse speed. The level of parallel execution of FPGAs allows an high exploitation concurrent execution in DNN model computation, compensating this lower value of frequency compared to alternatives. Then, FPGAs are able to handle the irregular parallelism in terms of bith-width for representation of network parameters and the DNN model sparsity introduced by some optimization methods through the possibility to design custom hardware logical units.

As a result, FPGAs represent the best compromise for accelerating DNN inference execution at the edge in terms of performance per unit power consumption, while maintaining high architectural flexibility and a small area occupancy. For ease of reading, the main advantages and disadvantages of FPGA-based accelerators have been summarized in Table 2.5 [40].

Table 2.5: Advantages and disadvantages of FPGA-based DNN inference accelerators.

FPGA DNN inference accelerator pros	**FPGA DNN inference accelerator cons**
High speed and high energy efficiency: FPGA-based DNN inference accelerators achieve high speed and energy efficiency.	*High time cost for reconfiguration*: Re-configurability of FPGAs comes at cost of time spent configure the best possible architecture for accelerating DNN processing.
High parallelism: FPGAs show a high level of parallel execution that can used to accelerate DL inference processing exploiting the high level of concurrency of DNN model computation through the development of highly parallel computing paradigms.	*High programming difficulty*: Re-configurable computing requires hardware level programming, which is usually much more complex than traditional high-level software programming languages.
High re-configurability: The ability to be configured multiple times allows FPGAs to perform complex tasks in optimally optimizing the design for considered ML application.	*Low adaptability*: The optimized configuration at the architectural level for FPGAs usually restricts the range of possible ML applications on which it can be used.

2.4 Internal Architecture of FPGAs

Field Programmable Gate Array (FPGA) is a type of re-programmable integrated circuit designed to enable the implementation of architectures containing custom logic units optimized for the application at hand.

At the architecture level, the building blocks of any FPGA are organized according to a grid of *configurable logic blocks* (CLBs) connected to each other through configurable interconnecting wires, with the function of implementing the logical operations for the considered task, and where the configurable connections decide how the different CLBs are connected to each other. The amount of CLBs available on FPGAs determines the number of logical units implementable on its area, thus also determining how complex a design implemented on that processing unit can be.

The FPGA internal architecture can be programmed to perform different operations in different parts within the same clock cycle. Thus, such hardware platforms are considered parallel by nature. In addition, within each FPGA structure there are also *I/O blocks* to exchange information with the external environment. For a more detailed overview of a standard FPGA internal structure, see Figure 2.16.

More in detail, each CLB consists of two basic components, which are the *look-up tables* (LUTs) and the *flip-flops* (FFs). While the former are registers used to store single bits of data, the latter are truth tables capable of implementing any combinatorial logic. Thus, in FPGAs it is possible to customize each LUT to implement any logical function.

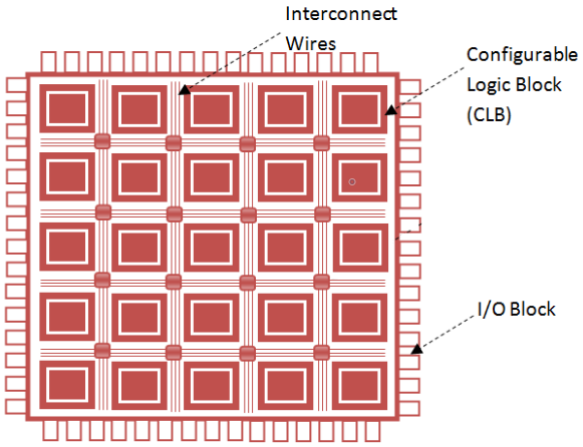

Figure 2.16: Internal architecture of a FPGA processing unit. Source [41]

Moreover, within any FPGA structure there is a set of *block RAM* (BRAM) areas, which cannot be configured for other functionality.

Through the I/O blocks, FPGAs have the possibility to interface with an external memory, which is usually necessary in order to contain all the data needed for the processing of the application under consideration, given the small size of the internal one.

In conclusion, due to their high degree of flexibility and parallel execution, FPGAs represent the optimal choice with respect to the execution of algorithms with a high level of concurrency, such as DL algorithms, where the execution of the MACs required for their processing can be highly parallelized. A further advantage of FPGAs for DNN model implementation is that in such platforms are implemented a certain quantity of *DSPs*, which are logic units capable of executing the MAC operation much more efficiently than the ALU units implemented on the LUTs.

2.5 Roofline Model Analysis for DNN Inference Accelerators

In the field of embedded applications, the throughput performance of any DNN inference accelerator is limited either by *computational* and *off-chip memory constrains*, which are due to the amount of internal PEs and external memory bandwidth, respectively. While the number of on-chip PEs determines the maximum number of MAC operations that can be executed simultaneously, the off-chip memory bandwidth limits the rate at which activations and weights can be read/written from/to the external memory and delivered to the internal PEs.

Specifically a hardware DNN inference accelerator is said *memory bounded*, when its performance is not limited by the external memory bandwidth, which therefore is unable to provide all the required data to be supplied to all the PEs for parallel execution, or *computation bounded*, in the case when all available on-chip PEs work simultaneously [42].

To analyse in detail the performance of any DNN inference accelerator, the *roofline model* has been introduced in [43].

Figure 2.17 shows a typical example of such a roofline model, with indicated both the *computational* and *bandwidth roofs*, where each point in the diagram represents a possible design of a hardware DNN inference accelerator.

More in detail, it proposes a method for evaluating the hardware accelerator performance (i.e throughput) taking into account the off-chip memory bandwidth constrains and the limited

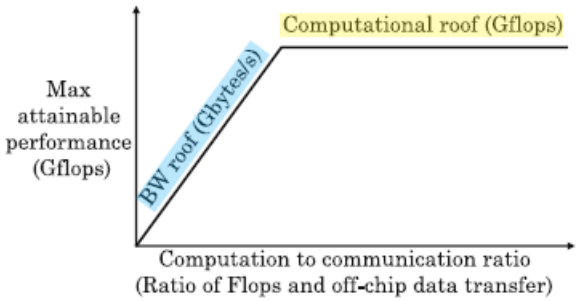

Figure 2.17: Illustration of a typical roofline model. Source [42]

amount of PEs available on the selected processing. In roofline model, the horizontal axes shows the *computation to communication* (CTC) *ratio*, which is a parameter used to measure the arithmetic intensity, i.e. the number of MACs that is possible to perform per off-chip memory access, evaluated in *#MACs/memory data access*, and the vertical axes indicates the maximum throughput achievable by the system, evaluated in *#MACs/DNN inference iteration*. The ratio $\frac{Y}{X}$ between any pair of coordinates of the roofline model is equal to the off-chip memory bandwidth of the selected hardware accelerator.

Looking at the Figure 2.18, it is possible to distinguish two distinct regions that make up the roofline model, which are:

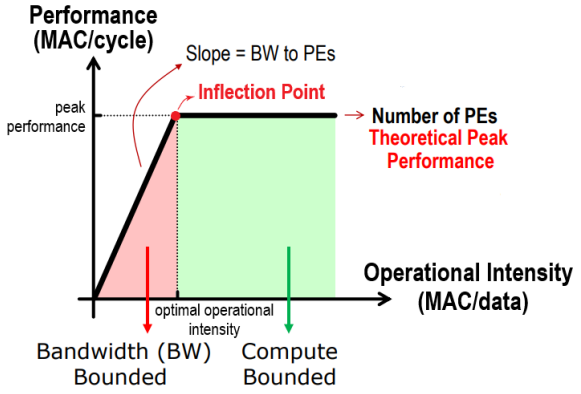

Figure 2.18: A detailed illustration of a typical roofline model. Source [43]

1. *Bandwidth roof bounded area* means that the design is *memory bounded*, and thus the performance is limited by the off-chip memory bandwidth of the selected hardware platform. The processing unit would be able to perform more MACs than it actually does, since the PEs required to do so are available, but due to the limited off-chip memory bandwidth, it can not provide enough data to be processed.

2. *Computational roof bounded area* means that the design is *computational bounded*, and thus the performance is limited by the amount of PEs available on the selected hardware platform. The number of MACs in execution is the maximum possible with available PEs, and thus the limited amount of PEs prevents further performance improvement.

1-2. The meeting point between the computational and bandwidth roof lines, referred to as *inflection point*, means that the off-chip memory bandwidth is able to provide enough data to perform all the MACs with the PEs available in the selected hardware platform.

As a result, the area bounded by computational and bandwidth roof line contains all possible designs for a hardware DNN inference accelerator that meet the PEs and off-chip memory bandwidth constraints of the hardware platform selected for implementing the DNN model under consideration. More in detail, the performance of the designed hardware accelerator is limited by the off-chip memory bandwidth when the arithmetic intensity is to the left of the inflection point, while when it is to the right is limited by the number of PEs that can work simultaneously.

Equation (2.1) expresses the maximum achievable throughput for any hardware DNN inference accelerator design, that corresponds to the minimum between the number of available PEs and the off-chip memory bandwidth.

$$Maximum\ attainable\ performance = min\ \{Computational\ roof, CTC \times BW\} \tag{2.1}$$

As a result, a higher arithmetic intensity corresponds to both a better data reuse (since it decrease the off-chip memory bandwidth requirements), or an increasing in memory bandwidth due to the application of some optimization methods.

Chapter 3

FPGA-Based DNN Inference Accelerators

In the following sections an in-depth analysis is carried out regarding FPGA-based DNN inference accelerators and the most important metrics used to analyze their performance, motivating the choice of this hardware platform for the implementation of complex DNNs through a roofline model analysis.

Subsequently, the main hardware/software-level optimization techniques used to improve the execution of DNN inference at the edge are detailed, focusing mainly on *hardware-oriented optimization methods for approximate computation*, being those that have shown the most promising results both in terms of speed and energy efficiency improvements, at the expense of a certain loss in accuracy.

3.1 Overview of Accelerating DNN Inference at the Edge

As discussed in Chapter 2, the choice of the appropriate processing unit for accelerating DNN inference is of paramount importance to undergo the latency/energy efficiency constraints imposed by edge ML applications.

In this context, if for the implementation of uncompressed DNNs the GPUs seems to be the only possible choice for achieving an high speed, the high power consumption avoid its integration in embedded devices. The development of *hardware/software-level optimization methods* supported by FPGAs capable of reducing the model complexity in terms of both computational and memory requirements has determined a transition towards re-programmable solutions for edge DNN inference processing.

An overview of the main hardware/software-level optimization techniques adopted to efficiently perform the DNN inference processing on FPGAs is shown in Figure 3.1.

Referring to Figure 3.1, while on the horizontal axis are shown the various abstraction levels within the ANNs on which it is possible to make such optimizations, starting from the general techniques *not related* to *DNNs*, passing to those at the level of *neurons* and *parameters* of each layer and those at the *layer* level, up to the methods at *network* level, on the vertical axis are identified three different software/hardware levels to which it is possible to work, which specifically are the *data-path*, *memory* and *operations scheduling levels*.

More in detail, to accelerate the DL algorithm inference, considerable interest falls on *approximating computation techniques* (circled in red in Figure 3.1) both at the level of *model compression* and *data quantization*, resulting to be the most effective ones in terms of both speed and energy efficiency.

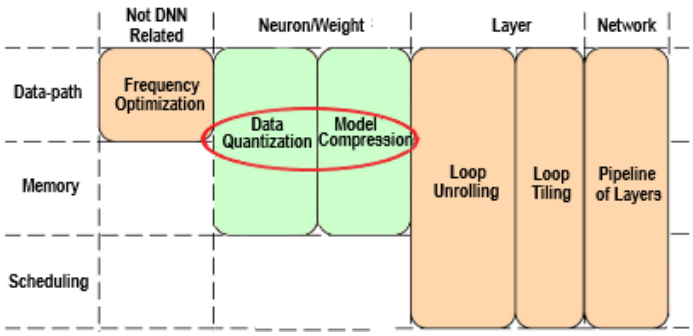

Figure 3.1: Overview about the hardware/software optimization techniques for improving DNN inference acceleration on FPGAs. Source [44]

Thanks to the increase in performance brought by the application of such approximating techniques, it is possible to mitigate the fact that FPGAs work at a lower working frequency than the alternatives, and therefore despite this fact implies a lower power consumption, it could be problematic for low latency ML applications. The high level of parallelism of FPGAs allows to exploit the inherent internal concurrency in DL algorithms execution increased by approximating algorithm application, resulting in a speed improvement.

As a result, provided some loss of accuracy is accepted, the DNN processing can be accelerated and energy efficiently improved by reducing DNN model complexity in terms of both the amount of computational and memory resources required per DNN inference iteration, facilitating the DL algorithm implementation on systems with a limited amount of resources.

For these reasons, over time FPGAs have proven to be superior for the hardware implementations of DL algorithms compared to alternatives, both in terms of speed and energy efficiency, while maintaining a low consumption of hardware computational resources and a small design cost, making it the optimal choice for the development of hardware accelerators for embedded applications.

3.2 Custom Hardware Design: Why FPGA-based DNN Inference Accelerators

Through an analysis of the roofline model introduced in Section 2.5 regarding the different processing units capable of accelerating the DDN inference processing, it is possible to understand why DNN model implementations on FPGAs, combined with the application of approximating methods, outperform the corresponding uncompressed GPU-based alternatives in terms of throughput, while maintaining a low design cost and high re-configurabilty compared to ASICs.

In Figure 3.2 is shown a comparison of the roofline models relative to the development of a hardware inference accelerator based on different processing units for the same DNN model. While for GPPs working in 32-bit floating point precision it is evident that GPUs outperform CPUs in terms of maximum performance due to both their higher level of parallel execution and optimized floating point computations, for FPGAs and ASICs supporting the fixed-point weight and activation representation with reduced precision, the latter outperforms the former. As result, it is evident that, compared to ASICs and FPGAs, GPUs show a significantly lower throughput.

Specifically, GPU internal computations are performed on a fixed architecture in a complex data type at high precision, requiring an high execution time and energy consumption. As a result, while it allows for a high level of accuracy, the maximum throughput achievable by GPUs is low. Unlike, ASICs and FPGAs, thanks to the support of reduced precision fixed-point data types, the ability of designing custom architecture with optimized data flows able to maximize the

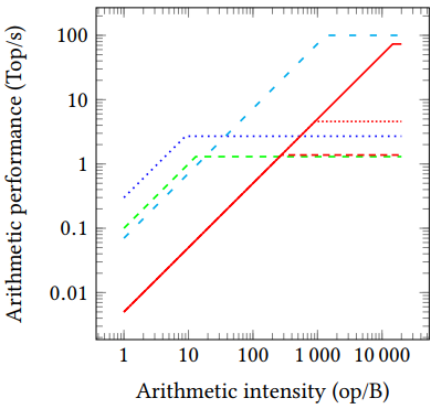

Figure 3.2: Comparison of roofline models for hardware DNN inference accelerators based on different processing units: CPU (– – –) and GPU (· · ·) with parameters in single precision floating-point (fp32) format, TPU (ASIC) (– – –) with parameters in 8-bit fixed-point (fixed8) format, and FPGA with parameters in 16-bit fixed-point (fixed16) format (– – –), 8-bit fixed-point (fixed8) format (· · ·) and 1-bit (———) fixed-point (fixed1) format. Source [45]

data reuse in the lower memory hierarchy levels and to the possibility of applying a wide range of optimization methods to approximate the internal computations, are capable of increasing the throughput of the hardware DNN inference accelerator.

As discussed in Section 2.5, for any hardware DNN inference accelerator the maximum throughput happens when all the PEs work simultaneously. Specifically, it is reached only if the hardware accelerator performance is not limited by the off-chip memory bandwidth. It is possible to increase the system performance through both a reduction in the precision of network parameters and the compression of the DNN model in terms of number of operations. While the former allows to improve both the use of the on-chip memory reducing off-chip memory accesses, it also improves the off-chip memory bandwidth, since in the same amount of moved data will be contained more information, the latter reduces the number of data transfers from memory to PEs, and so the number of MACs required per DNN inference iteration. Moreover, smaller compressed networks with lower-precision data to process allow for an increase in the amount of PEs that can be implemented in the same area, increasing the parallelism more.

In addition, approximating methods allow also for an increase in energy efficiency, mainly reducing the number of off-chip memory accesses increasing the data reuse, due to a decreasing in the number and in the precision of MACs per DNN inference processing.

Thus, for edge ML applications that tolerate a certain degradation in the precision of the results produced, the application of approximating methods on DNN model based on reduced precision and network compression allows to greatly improve the throughput and energy efficiency of the FPGA-based DNN inference accelerator.

3.3 FPGA-based DNN Inference Accelerator Architecture

As mentioned before, FPGA architectural flexibility and re-configurability allows to apply several optimization techniques in order to find the optimal solution, making them the best processing unit to accelerate the execution of the DNN inference at the edge. The typical structure of a FPGA-based DNN inference accelerator is shown in Figure 3.3.

More in detail, it consists of several components, which are an on-chip SRAM and an off-chip DRAM memory, an array of PEs, and the on-chip/off-chip interconnections used for data

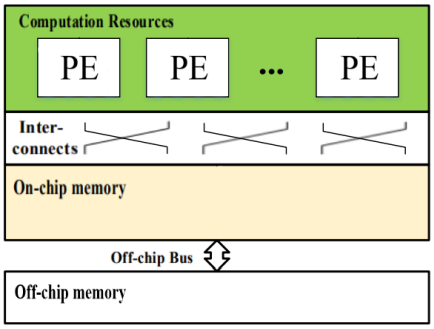

Figure 3.3: Internal structure of a FPGA-based DNN inference accelerator. Source [46]

communication between PEs and on-chip and off-chip memory, respectively.

The internal memory is able to contain all network parameters to feed all PEs, but usually it is not enough to store all the parameters required for a DNN inference iteration. To have an idea about the orders of magnitude, it is possible to estimate that performing a single DNN inference iteration requires an amount of weights and activations from 100 MB to 1000 MB, while, the largest available FPGA contains an internal SRAM of only 50 MB [42]. This problem is solved by interfacing to an external larger SDRAM memory, with a higher latency and energy cost per data access.

Though the computational capacity of FPGA-based DNN inference accelerator is high, thanks to the fact that implemented directly inside FPGA there are many DSP units allowing an efficient execution of MACs within CONV and FC layers, the interfacing between FPGA and SDRAM limits the performances of whole FPGA-DNN inference accelerator in terms of both speed and energy consumption.

To overcome this problem, the idea is to reduce as much as possible the number of accesses to external memory. Since FPGA are based on spatial architecture, it is possible to increase the data reuse through the development of energy efficient data flows and the introduction of an internal three levels memory hierarchy, which is composed of the external SDRAM containing the whole DNN model, an internal GB containing all the data required to feed all PEs, and the RFs within PEs. Thus, data fetched from the external memory are moved along different levels of the internal memory hierarchy, before being provided to PEs, where are used as many times as possible.

A detailed representation of the memory hierarchy for a generic FPGA-based DNN inference accelerator is shown in Figure 3.4.

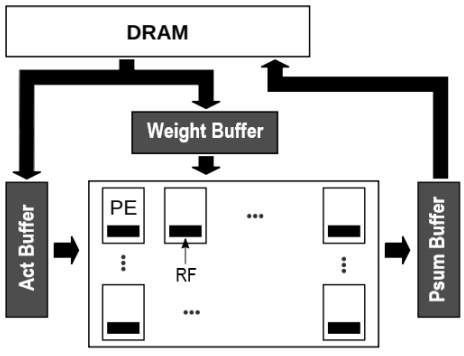

Figure 3.4: Memory hierarchy structure for a FPGA-based DNN inference accelerator. Source [32]

More in detail, the GB is in turn divided in three SRAM registers, which are the *activation buffer*, the *weight buffer*, and the *partial sum buffer* with a variable register parallelism. This allows the FPGA-based DNN inference accelerators to exploit the variable bit width of the different types of data. Thus, the architectural flexibility of FPGAs results to be feature of considerable interest for the realization of variable bit width hardware DNN inference accelerators.

3.4 Hardware Design and Evaluation Performance Metrics

In order to analyze FPGA-based DNN inference accelerator, it is important to introduce a set of useful metrics both at *DNN model* and *hardware accelerator* level, reported in Table 3.1.

Table 3.1: Main metrics used to evaluate DNN models (on the left) and FPGA-based DNN inference accelerators (on the right).

DNN model metrics	**FPGA-based DNN inference accelerator metrics**
*Accuracy [*pp*]* is a metric used to evaluate the quality relative to the results produced by a DNN model for a given task. It depends on the DNN model complexity, in terms of both number of MACs and network parameters, since the more complex it is, the higher the achievable accuracy. In addition, it is strictly related to the data type and precision chosen for the representation of the network parameters. The *type of DNN model network architecture*, intended as the number and the type of DNN layers is an important metric to understand how an DL algorithm elaborates the data to be processed. This is critical to determine what the best choices to make at the architecture and data flow design level. *Workload* represents the total amount of MACs to be performed in a DNN model per inference iteration. It is one of the main factors affecting the throughput/ latency and energy consumption of any DNN model.Workload is usually expressed as billions of operations [GMACs] per DNN inference iteration. The *memory requirements* of a DNN model depends on the amount of non-null weights and activations to be processed per DNN inference iteration. Having a DNN model with high memory requirements to store a very large number of parameters allows to achieve a high level of accuracy.	*Area efficiency* represents the amount of MACs and internal memory implementable on a FPGA with a fixed amount of hardware resources. The amount of on-chip memory represents the main critical resource in terms of area occupancy, even more than the computational logical units, and therefore optimizing its use allows not only to reduce power consumption and latency of FPGA-based DNN inference accelerator per inference iteration, but also to increase area efficiency. *Energy efficiency* is a metric adopted to express the the energy consumption of the FPGA-based DNN inference accelerator (measured in [J]) of the FPGA-based DNN inference accelerator required to perform all MACs and all memory accesses, to the weights and activations to be processed per DNN inference iteration. Energy efficiency related to computational contribution is expressed as the number of MACs processed within a unit of energy [GMACs/J], while that related contribution for data transfers from memory to PEs as the amount of energy required to read/write a byte [J/B]. *Throughput and latency* are metrics that related to the amount of PEs and off-chip memory bandwidth of the FPGA-based DNN inference accelerator. These metrics quantify the amount of time required to execute a DNN inference iteration and how often the results of a DNN inference processing are available as output. While throughput is measured as billions of MACs per second [GMACs/s], latency is measured in [s].

Thus, through these metrics, it is possible to evaluate the properties the DNN model and the corresponding FPGA-based DNN inference accelerator on which it will be implemented, to determine if the FPGA-based DNN inference accelerator implementing the DNN model is a feasible solution for the given task.

Specifically, they main one are the level of *accuracy* achievable by the DNN model on the results

produced per DNN inference iteration, to determine if the implemented DNN model is capable of performing the given task with the required quality, the *latency* and *throughput* achievable with the developed FPGA-based DNN inference accelerator, to determine if the speed of the hardware accelerator is sufficient for processing the given task, and the *energy efficiency* of the developed FPGA-based DNN inference accelerator, to determine if the amount of energy required to perform all the MACs and memory accesses required per a DNN inference iteration is compatible with the energy budget of the embedded device on which it is implemented.

Since they are mutually dependent on each other, during the design phase of any FPGA-based DNN inference accelerator for a specific DNN model it is critically important to understand how the change in one metric goes to affect the others. For example, it is possible to implement a FPGA-based accelerator with a high throughput and high energy efficiency, if not considering the influence on the DNN model accuracy, through the application of approximating computation methods. However, this implementation may not be usable for ML applications that require an high level of precision for the results produced. The maximum level of accuracy achievable by the DNN model is one of the main aspects to consider in the DL algorithm implementation on FPGAs for embedded applications, where the amount of PEs and on-chip memory is limited. As a result, it is necessary the presence of an external memory in which to store the data to be processed per DNN inference iteration, influencing speed and energy efficiency.

3.5 Design Methodology of FPGA-based DNN Inference Accelerators

The main goal of FPGA-based DNN inference accelerators is to achieve high speed (i.e. high throughput *th* and low latency *(*L)) and high energy efficiency (*Eff*), while maintaining an high level of accuracy. In the case of DNN inference processing in *embedded systems*, it is also of paramount importance to have a limited hardware resource consumption and a low total energy consumption E_{total}.

An analysis of the main factors affecting these metrics relate to a FPGA-based DNN inference accelerators is reported in the following (see Table 3.2 for a description of the design parameter symbols).

Table 3.2: List of design parameters of FPGA-based DNN inference accelerators. Source [44]

Symbol	*Description*	*Unit*
W	Workload per inference iteration	GOP
η	Utilization ratio of the PEs available in the FPGA	-
f	Working frequency of the FPGA	GHz
P	Number of PEs available in the FPGA	-
C	Concurrency factor of FPGA-based DNN inference accelerator	-
E_{static}	Static energy cost of the system per inference iteration	J
E_{op}	Average energy cost for each operation in inference iteration	J
N_{op}	Number of MACs per inference iteration	-
N_{access_mem}	Number of bytes accessed from on-chip/off-chip memory	B
E_{access_mem}	Energy for accessing each byte from on-chip/off-chip memory	J\B

Speed of a FPGA-based DNN inference accelerator is a key parameter for edge computing ML applications, which are usually characterized by both high *th* and low *L*. More in detail, they can be improved by acting on the following design parameters:

- *Throughput* (th) is a metric used to measure how often a FPGA-based DNN inference accelerator is able to perform a DNN inference iteration, and can be expressed as (3.1).

$$th = \frac{f \times P \times \eta}{W}, \tag{3.1}$$

  where f is the *working frequency* of the selected FPGA, η and P respectively the *utilization ratio of the available PEs* and the *number of PEs* on the FPGA, and W the *workload per DNN inference iteration*.

  It is possible to improve *th* in the following ways:

  1. *Increasing f*. The value of f is determined by the critical path of the FPGA-based inference DNN accelerator architecture, which is usually the path that connects the off-chip memory containing all the data to be processed during the DNN inference and the furthest PE able to perform a MAC operation. Thus, a possible solution to increase f may be to reduce the time required to perform the MAC operation by lowering the bit width and chose a simple data type for representing the DNN model parameters through *data quantization*. Memory bandwidth and type of memory (on-chip SRAM or off-chip DRAM) constitute another factor affecting *f*.
  2. *Increasing* η. The value of η quantifies the level of parallel execution of the FPGA-based DNN inferece accelerator, and thus provides an idea about the number of MACs that can be executed simultaneously per DNN inference iteration. It depends on both the amount of PEs available within the selected FPGA and the level of parallel execution possible of the DL algorithm. The off-chip memory bandwidth can be a limiting factor for η, as it may limit the number of PEs running at the same time by not providing enough data to process. Thus, an efficient memory management involving on-chip memory becomes critical to exploit the inherent parallelism of DNN model in order to increase the η value.
  3. *Reducing W*. The number of MACs required per DNN inference iteration defines the value of *W*. Thus, minimizing it means reducing the number of MACs necessary for each DNN inference iteration processing. To achieve this result is possible to apply the *network pruning* or to exploit the *ReLU statitics* to increase the level of sparsity of the DNN model.
  4. *Increasing P*. Since the amount of hardware resources available on any FPGA is fixed, the value of P is also limited. A possible solution to increase P is through the application of the *data quantization*, which provides the possibility of translating a decrease in data precision both in a reduction of the size of each MAC operator, thus making it possible to increase the number of PEs that can be implemented in the same amount of hardware area.

- *Latency* (L) is a metric used to measure the time between the arrival of inputs to be processed by FPGA-based DNN inference accelerator and the generation of the corresponding output. It is a key parameter for edge computing DL applications, which are usually characterized by low latency requirements. Since most of FPGA-based inference accelerators can inference one or multiple input samples in parallel at a time, the value of L can be derived from the *th* parameter through (3.2).

$$L = \frac{C}{th}, \tag{3.2}$$

where C represents the *concurrency factor* of the FPGA-based DNN inference accelerator, that is a indicator on the number of inference processed in parallel. As a result, since these two quantities are inversely proportional, the increase of th allows to reduce L (and vice-versa), and consequently similar reasoning to those made to increase th can be made also to reduce L.

Energy efficiency, is a metric used to quantify the amount of energy required by a FPGA-based DNN inference accelerator to process all the MACs and perform all the memory accesses per DNN inference iteration. It is a key parameter for edge computing applications, which are usually characterized by a limited energy budget, and can be expressed as (3.3).

$$\mathit{Eff} = \frac{W}{E_{\text{total}}}, \tag{3.3}$$

where E_{total} is the total energy cost per DNN inference iteration, and is expressed by (3.4).

If W is fixed, increasing Eff means reducing E_{total} required to process a DNN inference iteration.

$$E_{\text{total}} = N_{\text{op}} \times E_{\text{op}} + E_{\text{access\_SRAM}} \times N_{\text{access\_SRAM}} + E_{\text{access\_DRAM}} \times N_{\text{access\_DRAM}} + E_{\text{static}} \tag{3.4}$$

where N_{op} is the number of operations per DNN inference iteration, and is expressed by (3.4).

The first term in (3.4) is the dynamic energy required to perform all required MACs per DNN inference iteration, the second and the third terms, respectively, the dynamic energy contribution required to make memory accesses to on-chip SRAM and off-chip DRAM for reading/writing the parameters to be processed per DNN inference iteration, and the fourth one the static energy contribution of the system.

Thus, the total energy consumption of a FPGA-based DNN accelerator per DNN inference iteration E_{total} depends on both the energy required for the *memory accesses* to read the data to be processed per DNN inference iteration through the PE array, and the energy required for the *computational processing* of MAC operations.

- In terms of *memory accesses*, it is possible to reduce E_{total}, by working on the second and third terms of (3.4) in the following ways:

  1. *Reducing* $E_{\text{access\_memory}}$. The *energy for accessing a byte in memory* ($E_{\text{access\_memory}}$) is the dominant contribution in terms of energy consumption, and depends both on whether the data access is made from the off-chip or the on-chip memory. More in detail, as shown in Table 2.1, if the memory read operation is performed from the external one, it consumes much more energy than if it is made from the FPGA internal one. In addition, as shown in Figure 2.8, the energy cost of accessing data contained in different levels of the on-chip memory hierarchy can vary greatly. To reduce $E_{\text{access\_memory}}$ one possibility is to avoid off-chip memory accesses as much as possible by maximizing the reuse of data present at lower levels of the on-chip memory hierarchy for multiple MAC operations, through the design of efficient data flows (see Section 2.2). Since for FPGA-based DNN accelerators the amount of internal SRAM is hardly enough for a DNN inference iteration processing, it is usually not possible to completely avoid access to the external DRAM.

2. *Reducing* $N_{\text{access\_memory}}$. The *number of bytes accessed from memory* ($N_{\text{access\_memory}}$), depending on the internal structure of the DNN model, can be reduced through the application of some optimization methods, such as *data quantization*, to decrease the bit width of the read data from memory to be processed for DNN inference processing, *network pruning* and *ReLU statistics* to increase the level of sparsity of the DNN model so as to reduce the number of accesses in memory to read the weights and activations.

- In terms of *computational processing*, it is possible to improve E_{total}, by working on the first term of (3.4) through the *data quantization*, decreasing the bit width and simplifying the data format to decrease E_{op}, and the *network pruning*, skipping the MACs that involve a multiplication with a zero value to reduce N_{op}.

- The E_{static} term represents an energy contribution difficult to reduce, but usually it is not a problem, since it is small compared the other ones.

3.6 DNN Inference Acceleration on FPGAs

Over time, the need to obtain an ever-increasing level of accuracy on the results produced has led to the development of ever deeper DNN models. As a result, the direct implementation of DNN models in embedded devices for edge computing may be difficult, since the amount of available computational and memory resources is limited and the DL applications to be executed are usually subject to strict latency and power consumption constraints, in addition to a low hardware resource budget.

While the hardware implementation of DL algorithms on FPGAs allows to greatly improve the DNN inference processing at the edge both in terms of speed and energy efficiency, mainly due to their flexibility and configurable architecture and high level of parallelism exploitation, the limited amount of resources available on them limits its fields of application for tasks requiring a high level of accuracy. The problem lies in the high demand in terms of both memory requirements (and therefore memory size, memory bandwidth and energy consumption for data movements especially between off-chip memory and PE array), and computational resources (and therefore amount of PEs and energy consumption required to perform the MACs in FC and CONV layers) per DNN inference iteration.

Since data movements dominate power consumption and limit FPGA-based DNN inference accelerator throughput, one solution is to reduce data transfers from off-chip memory to PEs through the development of computing paradigms with energy efficient data flows, capable of maximizing the reuse of the data at the lower level of on-chip memory hierarchic. Regarding these *architectural-level design improvements* of FPGA-based DNN inference accelerators described in Section 2.2, it is important to say that these methods does not result in any loss of system accuracy.

For ML applications where the desired level of accuracy is not too high, in order to improve the performance of any FPGA-based DNN inference accelerator in terms of speed and energy efficiency, it is possible to modify the internal structure of the DNN model during the design to reduce computational and storage complexity through the implementation of some *approximating computation techniques*, at the expense of some degradation in the final accuracy. In general, these approximation methods allow increasing the performance of FPGA-based accelerators by increasing the level of parallel execution and reducing the number of data transfers and the amount of MACs required per DNN inference iteration.

Since the application of these methods requires a high level of flexibility of the processing unit on which the DL algorithm is implemented, FPGAs represent the optimal choice. Thus, they result of paramount importance in the context of accelerating DNN inference on FPGAs, because they allow to increase both the speed and energy efficiency.

One important thing to add is that, unlike the architectural improvements that do not result in any loss of system accuracy, the application of these approximating computation methods can significantly affect the level of accuracy of the FPGA-based DNN inference accelerator. As a result, it is important to understand on a case-by-case basis whether the amount of accuracy lost on the results produced through the application of such hardware-oriented optimization methods for approximate computation is acceptable for the ML application under consideration.

Taking into account the design metrics related to FPGA-based DNN inference accelerators introduced in Section 3.5, a wide variety optimization methods have been developed over time, to improve both the speed and energy efficiency for DNN inference processing at the edge (see Figure 3.5).

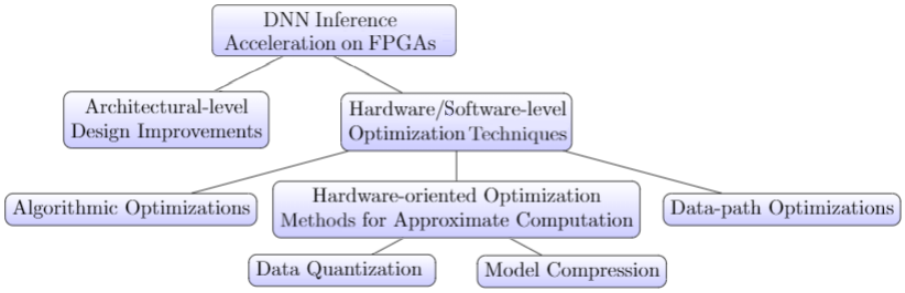

Figure 3.5: Classification of architectural-level design improvements and HW/SW-level optimizations methods to accelerate DNN inference processing on FPGAs.

More in detail, FPGA-based DNN inference accelerators support a wide range of *hardware/software-level optimization techniques*, which focus mainly on reducing the complexity of the DL algorithm to be implemented both in terms of both computational and memory requirements. The objective consists in improving the speed and the energy efficiency of the system by acting at the level of both *DL algorithm optimizations* to be implemented and of implemented *DNN model data-path optimizations*, or even through some *hardware-oriented optimization methods for approximate computation*, especially with regard to solutions based on processing units with re-programmable custom hardware with an high degree of architectural flexibility. Specifically, the ability of FPGA-based DNN inference accelerators to implement logical units optimized for the task at hand combined with the application of these approximation algorithms results in a substantial reduction of both computational and memory requirements for the DNN inference processing, which is the main goal for edge ML applications critical in terms of throughput/latency and energy consumption.

Particularly interesting are the latter, that although determine in a certain loss of accuracy, are the most effective one in terms of speed and energy efficiency improvements. More in detail, they are mainly based on changing the data type (both in terms of data format and bit width) of the weight and activations to be processed per DNN inference iteration, in order to reduce the complexity of operands and operators (i.e MACs) for storage/compute, or on increasing the level of sparsity of the DNN model network parameters, and on structural simplifications at the network level (and therefore with a lower number of neurons and connections).

Table 3.3 provides a classification of the idea on which are based the approximated computation methods applied on FPGA-based DNN inference accelerators.

Thus, to make feasible the implementation of complex DNN models on embedded devices, the main goal is to have as few parameters and operations as possible for DNN inference processing in the lowest precision that allows to achieve the correct result with an acceptable level of accuracy for the task at hand. Thus, by reducing the precision used for the representation of the weights and activations to be processed, the number of MACs for inference iteration and the DNN model size, it is possible to decrease both the computational and memory requirements for the execution

Table 3.3: Classification of hardware-oriented optimization methods for approximate computation.

Reduce precision of operands and operators	**Reduce number of operations and model size**
From floating-point to fixed-point	Sparsity exploitation
Reduction in the number of bits	Structural simplifications

of the DL algorithm under consideration, at the exepense of a certain loss in accuracy. As a result, accepting a small loss of precision on the results due to removing some MACs and lowering the precision of network parameters, it is possible to achieve significant improvements in both throughout/latency and energy consumption per DNN inference iteration, in addition to a lower hardware resource utilization.

More in detail, it is possible to classify *hardware-oriented optimization methods for approximate computation* into two main categories, which are the techniques oriented towards *reducing the precision* (in terms of both the data type and number of bit for the representation of DNN model parameters) *of weights/activations and MACs* through the *data quantization*, and the techniques oriented towards *reducing the number of MACs and the model size* through the *model compression* to increase the level of DNN model *sparsity* (see Figure 3.6).

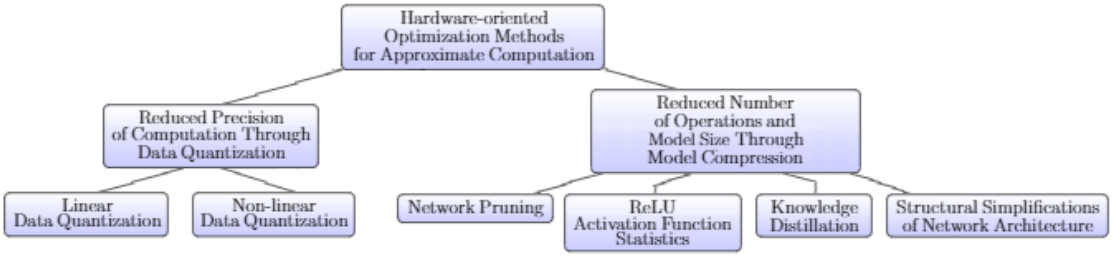

Figure 3.6: Classification of HW-oriented optimization methods for approximate computation.

The following sections provide a detailed overview about such optimization techniques for approximate computation, describing their main advantages and disadvantages.

3.6.1 Data Quantization to Reduce the Computation Precision

Regarding the class of approximating methods that can reduce the precision of operands (weights and activations) and operators (MACs) contained within the DNN model architecture, the main idea on which these techniques is the quantization of data.

In general, *data quantization* consists of mapping a set of continuous data onto a smaller range of discrete data, referred to as *quantization levels*, through the application of a certain *quantization function*, where the amount of bit selected to represent the quantized data determines the number of quantization levels. The difference between the quantized and original value is called *quantization error*, and must be as small as possible, in order to have an optimal quantization procedure. As a result, quantized data have a lower level of precision than the original values.

In DL processing, the default data type for representing weights and activations of a DNN model is the so-called *single-precision floating point format* (fp32) (see Figure 3.7). In fp32 the possible values of the number stay in the dynamic range that goes from 10^{-38} to 10^{+38}.

As DNN inference processing is robust in terms of accuracy on the results produced relative to a reduction in precision of the data to be processed, in FPGA-based DNN inference accelerators

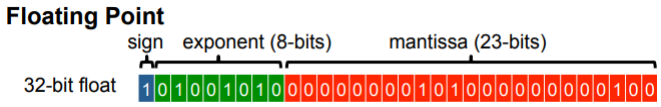

Figure 3.7: 32-bit (single-precision) floating-point data representation. Source [47]

it is possible to leverage the ability to implement hardware units at lower precision to achieve a better throughput/latency through an increasing of parallelization and a reduction of the off-chip memory bandwidth requirements, and a better energy efficiency reducing the number of off-chip memory access required per DNN inference iteration.

Data quantization allows to replace the fp32 data type with a lower bit width fixed-point data format, moving toward simpler data representation of weights and activations most beneficial for DNN inference processing in hardware. Any *N-bit fixed point* (fixedN) number in *two's complement* is represented by $(-1)^s \times m \times 2^f$, where s is the sign bit and m is the N-1 bit mantissa. f determines the position of the binary point among integer bit$I$ by fractional bit$F$, and consequently may be used as a *scaling factor*. The possibility of modifying the value of f makes the fixed-point format dynamic in terms of range of values and resolution it is possible to represent on the same number of bit. As a result, its dynamic range goes from -2^{I-1} to $2^{I-1} - 2^F$, with a resolution equal to 2^{-F}. Figure 3.8 illustrates the case of a number in 8-*bit fixed point data representation*, which is widely used for DNN inference processing for representing weights and activations.

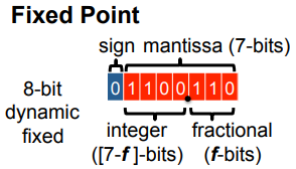

Figure 3.8: 8-bit dynamic-fixed point data representation. Source [47]

The optimal number of bit on which to represent the weights and activations to be processed during DNN inference processing depends on the requirements on the accuracy, speed and energy efficiency of the ML application under consideration. The more the precision for data representation is lowered, the more are the advantages in terms of speed and energy consumption saving, at the expense of a much higher loss of accuracy.

More in detail, on the same number of bits the floating-point dynamic range is much wider and with an higher resolution than the fixed-point one. As a result, after quantization some DNN parameters may be affected by some loss of precision, which in turns affects the system accuracy. This problem can be partially compensated by modifying the value of f that allows to have different dynamic ranges of values and resolutions. This feature is particularly useful for the representation of weights and activations for accelerating DNN inference processing. In fact, for DNNs not only the numerical range of values of different data types can be different, but it can also differ depending on the position and the type of the considered layer. The possibility in fixed-point to decide a variable dynamic range for both different types of network parameters and different layers allows to not lose too much accuracy after data quantization, and to reduce the memory bandwidth and optimizing the memory size requirements.It results to move from a *static quantization* for the whole DNN model towards a per-layer and per-parameter *dynamic quantization*.

In addition, while the implementation of floating point MACs requires an elevated resource consumption and results in a low speed and high power consumption, the fixed-point operations are traceable to operations between integers, and then scaled to effective value by multiplying the

obtained result with *f*. As a result, the cost in terms of hardware resource consumption is greatly reduced compared to that of the floating-point format, allowing also a faster and more energy efficient MAC execution.

The possibility of adopt a fixed-point format with lower precision for weights and activations is particularly interesting in the context of FPGA-based DNN inference accelerators for embedded applications, where a high level of accuracy is not required, and the main goal is to implement DNN models on a limited amount of hardware resources capable of running all the MACs required for the execution of the DL algorithm inference with low latency and small power consumption. Thus, the goal of the data quantization technique in the context of the FPGA-based DNN inference accelerator is to reduce the size of the DNN model to be implemented in hardware in terms of operand (weight and activation) and operator (MAC) precision in order to speed up the edge processing and achieve better energy efficiency for both computations and memory accesses, at the expense of some loss of model accuracy on the results produced.

More in detail, the main difference between the two types of data types is that, provided some loss of system accuracy is accepted by changing data representation, the fixed point format is much less expensive than the floating point one, both in terms of computational (and thus in time and energy consumption required to execute each operation) and memory requirements (and thus in memory size, memory bandwidth and latency/cost per memory data access).

Data quantization focus primarily on weights rather than activations, since the former affects the DNN model accuracy less than the latter. First, the influence of weights on memory requirements of the FPGA-based DNN inference accelerator are direct, while the apport of activation depends on the designed architecture and the chosen data flow. Second, while the weight quantization process is performed only once in the cloud after the training (because after determining the weight values of the DNN model for a given task to be loaded on the hardware platform in reduced precision, if the task remains the same they do not change their value over time) with the possibility of fine-tune them to recover percentage points of accuracy, the activations should be quantized at the edge at each DNN inference iteration, resulting in an high overhead in latency per DNN inference iteration.

For these reasons, although reducing the precision of activations is useful for improving both the speed and energy efficiency of any FPGA-based DNN inference accelerators, it involves a much higher loss of accuracy on the results produced by the DNN model than weight quantization.

A flow-chart related to the data quantization method with the possibility of fine-tuning the value of DNN model parameters is shown in Figure 3.9.

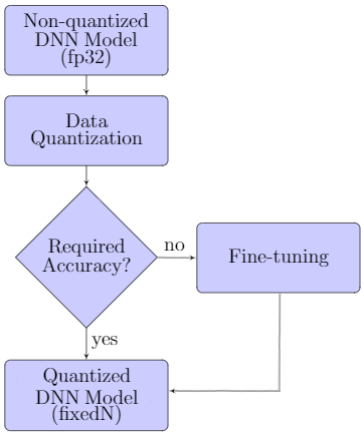

Figure 3.9: Flow-chart for performing data quantization and fine-tuning of network parameters.

To summarize, the application of the data quantization techniques on weights and activations of DNN models affects both the computation and memory. As for the former, while moving from fp32 to reduced fixed point data format for weights and activations decreases MAC complexity, improving the execution speed and energy consumption at operation level, it also reduces the hardware resource consumption. As for the latter, it results in a reduction in both the memory size, and so in the latency and the energy consumption per data access, and in off-chip memory bandwidth required for DNN inference processing. As a result, if the level of parallelization of the DL algorithm allows, the increasing the number of available PEs and the improving in the off-chip memory bandwidth results in an higher system performance, since the number of MACs performed simultaneously during each inference iteration increase significantly.

The architectural flexibility and reconfigurability of FPGAs makes them the optimal hardware platforms for implementing DNN models for accelerating the inference with a simpler and lower precision processing, allowing an increase in performance in terms of both throughput/latency due to higher computational speed, and energy efficiency, due to a reduction in the number of off-chip memory accesses, at the expense of a better use of the on-chip one.

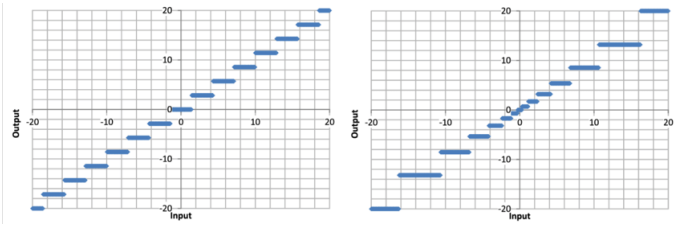

Figure 3.10: Linear (on the left) and logarithmic (on the right) quantizations. Source [48]

More in detail, data quantization techniques can be classified in two categories, which are *linear* and *non-linear quantization* methods (see Figure 3.10).

– *Linear quantization*: it is a data quantization technique characterized by an uniform spacing between different quantization levels. At the expense of a certain loss of accuracy of DNN model, it allows to improve the speed and energy efficiency of the corresponding FPGA-based DNN inference accelerator, as well as reducing area and memory occupancy of the DNN model. While regarding the computational part a simpler data type allows to make faster the MAC operation, for the memory requirement reduction is not sufficient to change only the data format, but it is necessary also a reduction in precision of the data format. It is important to add that, after linear quantizing the data, it is typically necessary that the internal precision of each MAC operator will be higher than the one used for weights and activations to be processed, to not lose too much accuracy due to possible overflow/underflow errors in the MAC execution (see Figure 3.11). Then, the precision of the MAC output is reduced to N-bit to maintain the parallelism uniform.

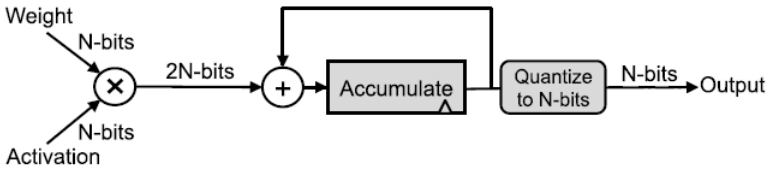

Figure 3.11: Precision in MAC operation. Source [5]

Beacuse of the different requirements in terms of bit width for weights and activations and

for the layers placed in different positions in the DNN model, with the aim to limiting the accuracy degradation it is also possible to perform a linear quantization depending on the type of parameter and on the position of the layer in the DNN model, referred to as *dynamic fixed-point data quantization*, where different layers are associated to different scaling factors. This is very useful to improve the speed and energy efficiency in FPGA-based DNN inference accelerators, since by exploiting the architectural flexibility of FPGAs in realizing custom logical units it is possible to implement arithmetic operations on different numbers of bits, reducing computational requirements. In addition, it improves also memory occupancy, at the cost of handling different parallelisms. Among linear quantizations, *extreme quantization* is a type that leads to reduce the bit width for representing network parameters to a extremely small number of bits, with associated a scaling factor. It allows to improve substantially the DNN inference processing in terms of speed and energy efficiency, at the expense of a greatly loss in accuracy. In particular, when weights and activations are quantized to a single bit, it is referred to as *binary quantization*. In BinaryConnects (BCs) only the weights are quantized to '−1' and '1', while maintaining activations on fp32 in order to perform MACs as additions/subtractions, depending on whether the value of the weight is positive or negative. *Binarized Neural Networks* (BNNs) quantize both weights and activations to '0' and '1', allowing to reduce each MAC to a XNOR. While extreme quantization accounts for both huge computational and memory improvements , the loss of accuracy compared to the fp32 DNN model may be very high, limiting the range of ML applications. To recover some accuracy, it is necessary to introduce several optimizations, such as using an appropriate scaling factor for the weights to restore the value to multiply or keeping the last layers of the DNN model in 32fp, since the last layers influence more the DNN model accuracy than the first ones. A further possibility to improve the accuracy is to add an additional bit for the quantized network parameter representation, denoted as *ternary quantization* in the so called *Ternary Nets* (TNs). More in detail, it is to add the value '0' to the possible values assumable by the weights. The introduction of the additional bit with zero value allows to make sparser the DNN model in terms of weights, increasing the hardware resoruce consumption while reducing computation and memory requirements, with the aim to accelerate the DNN inference.

– *Logarithmic quantization*: it is a data quantization technique characterized by a non-uniform spacing between the different quantization levels, adopted to follow better the non-uniform distribution of network parameters to reduce the loss of accuracy. More in detail, the network parameters are quantized in power of 2 with associated a scaling factor. Thus, it allows to increase the range of values that can be represented on the same number of bits, paying in resolution. Up to now, although it is possible to apply it also to activations, logarithmic quantization is only for weights, since for them in order to maintain high the model accuracy is more important the range of values than the resolution. Thus, by quantizing only the weights through a logarithmic function, it is possible to substitute the multiplication within each MAC with a *binary shifter*, since the weighted sum between weights w_i and activations x_i can be expressed as (3.5).

$$w_i \times x_i = \sum_{i=0}^{N} w_i \times x_i = \sum_{i=0}^{N} w_i \times 2^{\tilde{x_i}} = \sum_{i=0}^{N} w_i \ll \tilde{x_i} \tag{3.5}$$

where $\tilde{x_i} = Int(\log_2(x_i))$.

Thus, replacing the multiplication with a shifter allows to achieve improvements both in terms of speed and energy efficiency, reducing also the hardware consumption. Moreover, in FPGAs, if the shifting to perform remains constant, it can be implemented simply through a routing operation, determining additional improvement in terms of speed and resource

consumption. Although there are a wide range of non-linear quantizations methods allowing to reduce even more the quantization error, for FPGA- based DNN inference accelerators the logarithmic quantization is particularly interesting for the limited amount of hardware resources that it requires to be implemented.

A comparison for standard CNN models in terms of *top1* and *top5* accuracy loss in ImageNet for the same classification task of the linear and non-linear data quantization methods with different bit widths for representation of weights and activations with respect to corresponding accuracy obtained in fp32 is shown in Figure 3.12.

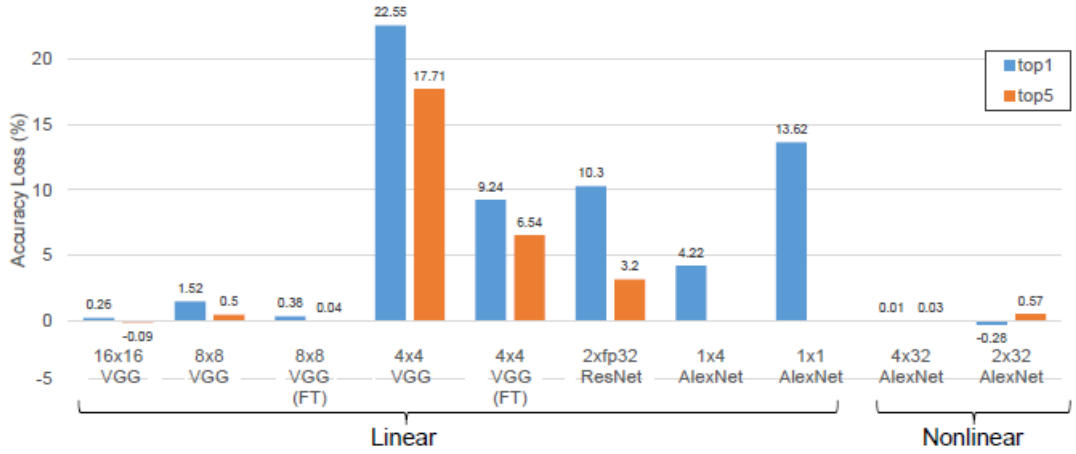

Figure 3.12: Comparison of linear and logarithmic data quantization techniques in terms of accuracy loss in ImageNet. The selected quantization precision is indicated as (weight bit width × activation bit width), while FT label means that weights was fine-tuned after quantization. Source [44]

More in detail, each quantized configuration is expressed as the number of bit chosen to represent the weights and activations, respectively, and the *FT* label indicates that a fine-tuning operation has been performed only on the weights (which is the so-called *post-trainign quantization*) in order to recover at least part of the accuracy loss after data quantization.

Looking at the Figure 3.12 it is possible to say that regarding to linear quantization, the optimal bit width determining a negligible accuracy loss relative to the quantization of both weights and activations in fixed-point is 8. It determines a negligible accuracy loss also without fine-tuning, while results a high hardware resource consumption saving, reducing also the computational and memory requirements for DNN inference processing. By contrast, non-linear quantization applied only on the weights enlarges their range of re-presentable values, yielding to better results in terms of FPGA- based DNN inference accelerator accuracy loss by quantizing them to a very small number of bits, provided the activations remain in fp32 data type.

Reducing the Size of Computational Units

In data quantization, the selected data type and number of bits for the weights and activations to be processed by the MACs has an impact both on the accuracy of the DNN model and on the performance of the FPGA-based DNN inference accelerators in terms of speed and energy efficiency.

Specifically, the hardware implementation of reduced precision computational units (i.e. from fp32 to fixed-point with a reduced bit width) working on simpler fixed-point data format through data quantization leads to an increase in both the speed of MAC execution and in the number of

PEs that can be implemented in the same area, with also an improvement of the off-chip memory bandwidth. While the former results in an overall acceleration of FPGA-based DNN inference accelerator in speed (i.e. latency and throughput).due to an increase in the working frequency, the latter increases the degree of parallel execution of the DNN inference processing, increasing the number of concurrent working MACs. Thus, in FPGA-based DNN inference accelerators data quantization is largely adopted.

For FPGA-based DNN inference accelerators able to handle variable bit widths, it has been shown that in order to optimize the DNN inference processing in terms of accuracy and computational/memory requirements, the number of bit required for representing weights and activations of different layers is different. As a result, depending on the type of layer in the DNN model, in FPGAs can be designed computational units with an optimized number of bits to both accelerate DNN inference processing and decrease hardware resource consumption.

Table 3.4 presents a comparison with respect to the FPGA hardware resource consumption required to implement a DNN model with the network parameters represented in different data formats and with different precisions. Specifically, the two types of FGPAs considered on which to base the development of the hardware accelerator are able to implement the MAC operator or through LUTs and FFs to realize a multiplier and an adder (i.e. through *logical resources*) or through the implementation of the multiply&add function through DSP units.

Table 3.4: FPGA resource consumption for implementing the MAC operation with different data formats and precisions for logical resource (on the left) and DSP (on the right) implementation.

FPGA (Logic)				
	multiplier		adder	
	LUT	FF	LUT	FF
fp32	708	858	430	749
fp16	221	303	211	337
fixed32	1112	1143	32	32
fixed16	289	301	16	16
fixed8	75	80	8	8
fixed4	17	20	4	4

FPGA (DSP)			
	MAC		
	LUT	FF	DSP
fp32	800	1284	2
fp16	451	686	1
fixed32	111	64	4
fixed16	0	301	1
fixed8	0	0	1
fixed4	17	20	0

Looking at the Table 3.4 [44], it turns out that relative to the implementation of MACs based on logical resources, a homogeneous quantization of network parameters from fp32 to fixed8 data format allows for reducing greatly the consumption of hardware resources available on the FPGA, so as the one relative to the implementation based on DSPs. A further reduction in bit width relatively to fixed-point representation of weights and activations to be processed in DNN inference would result in an additional decrease in terms of the size of the computational units and hardware resource consumption, but at the same time would lead to a loss in DNN model accuracy that is not acceptable for most ML applications (refer to Figure 3.12).

An additional consideration regarding the MAC implementation with DSPs is that the implementation of the multiply&add function on a number of bit equal or less to 8 involves or just 1 DSP, or the switching to the implementation of the MAC operation based on multipliers and adders on the FPGA logical resources. Thus, when the precision chosen for the quantized network parameters is lower than the one of DSP unit, it would result in an inefficient MAC implementation, and so it is shifted to the FPGA logical resources. Since usually DSPs are the most critical units on FPGAs in terms of number, it allows to increment the parallelism of the system increasing the number of MACs. However, quantizing below fixed 8 bit increases substantially the accuracy loss of the FPGA-based DNN inference accelerator, as mentioned before.

Tools to Optimize Data Quantization

In low latency and low power edge computing, data quantization is an important approximation method to improve both the speed and energy efficiency of a FPGA-based DNN inference accelerators. However, the standard *post-training quantization* consisting to quantize the fp32 network parameters of an already trained DNN model in a lower precision fixed-point data representation on a constant number of bits for the whole DNN model may result not only in a high loss of accuracy not acceptable for many ML applications, but also in a bad management of the computational and storage resources, limiting the performance of the designed FPGA-based DNN inference accelerator.

Some improvements in terms of model accuracy with respect to the fixed bit width *homogeneous quantization* can be achieved with *heterogeneous quantization* by selecting a variable number of quantization bits depending on the layer position in the DNN model and on the network parameter type. Thus, this type of data quantization allows to improve the accuracy, resulting of particularly interest for FPGA-based DNN inference accelerators, that due to their ability to implement custom logical units in their architecture support variable bit widths.

Over time, with the aim to achieve even more improvements, the focus has moved to the development of alternative approaches to reduce the precision of weights and activations pre-training and on variable precision, referred to as *heterogeneous quantization-aware training* methods. More in detail, it is performed a type of quantization depending on the impact of each element to be quantized on the final model accuracy, optimizing also the DNN model size and performance in terms of speed and energy efficiency.

However, finding manually the optimal configuration for the DNN model adopting this type of quantization method among all possible ones is a challenging task, since the more the number of layers of the DNN model increases, the wider the possible solution space becomes. More in detail, in order to find the optimal heterogeneous quantization configuration among all possible ones in terms of reducing hardware resource and energy consumption without compromising model accuracy, one should look not only how each quantized layer affects the value of the loss function in terms of resulting model accuracy, but also how it affects the systems in terms of hardware resources and power consumption reduction.

A methodology to automatically find the optimal heterogeneous quantized configuration for a given DNN model in terms of both resource consumption, latency and power consumption has been proposed in [49] through the use of some tools able to support the development of the data quantization process. It is developed through the use of the quantization extended versions of the Keras framework, called *QKeras* and *AutoQKeras*, specifically designed for simplifying the implementation of quantized DNNs starting from the corresponding Keras model. Specifically, Keras is a tool capable of supporting the development and the training of DNN models. It allows to estimate in advance the DNN model performance and the total amount of MACs and memory accesses required per DNN inference iteration. Knowing these results before the DL algorithm hardware implementation allows to select the optimal quantization configuration for the network parameters to achieve a certain level of performance for the FPGA-based DNN inference accelerator to be designed.

More in detail, the QKeras tool support the creation and the training of any heterogeneously quantizated version of the considered Keras DNN model by replacing each of its layers with quantized counterparts. It contains a set of functions to perform the transformation from non-quantized Keras to quantized QKeras DNN model. Specifically, it is necessary to specify the *QKeras quantizer*, which defines the type of quantization and the bit width on which to represent the weights and activations of the quantized QKeras model layer by layer.

Accordingly, the objective is to design a method to perform automatically the heterogeneous quantization-aware training, while minimizing hardware resource and energy consumption and simultaneously maximizing model accuracy. *QTools* is a tool for estimating the model size and

energy consumption of quantized QKeras models. Specifically, by providing the selected data type for the parameters and activations of each layer (in terms of data format, quantization type and bit width), it allows to estimate the corresponding per-layer energy consumption due to the memory accesses and MAC operations of the considered quantized QKeras model. The measurements produced by QTools for each QKeras model layer provide an insight into how to fine-tune the quantized heterogeneous model configuration considering both accuracy and consumption of energy and hardware resources. Thus, based on these results, the idea is to automate the procedure to find the optimal heterogeneous quantization configuration for a given Keras DNN model under a set of resource constraints imposed by edge application, trading-off an acceptable loss of model accuracy for a given reduction in model size or energy consumption.

More in detail, since for edge application the desired level of accuracy and the amount of hardware resources available for DL algorithm implementation are known in advance, by estimating with QTools the model size and energy consumption of a QKeras model layer by layer, it is possible to define the so-called *forgiving factor*. It represents a loss function different form the standard one, that takes into account both the cost in terms of hardware resource and energy, as well as changes in accuracy. Thus, knowing the value of the forgiving factor of a QKeras model, it is possible to worsen the value of a given model metric, such as model accuracy, so as to improve the model in some other metric significantly, like in model size or energy consumption.

As a result, through the calculation of the forgiving factor and the estimations of the QKeras model size and energy consumption of its layers provided by QTools, using the AutoQKeras tool, an automatic procedure of per-layer/per-parameter quantization-aware training and fine-tuning can be defined to find the optimal quantization configuration for the task at hand. Specifically, the selection of the optimal quantization configuration is treated as an hyper-parameter of the QKeras DNN model to be iteratively tuned.

As quantizing the whole Keras model at once greatly increases the space of possible solutions, there are two methods to perform the automatic heterogeneous quantization in AutoKeras, which are either quantizing groups of layers with the same quantization type, or quantizing blocks of layers by going in sequence, or quantizing blocks of layers with higher energy consumption first. It also provides the ability to exclude layers from the quantization process that if quantized would affect the model accuracy too much (the latter layers of the network usually have a greater impact on the accuracy value of the model than the former).

Thus, staying below the acceptable accuracy loss value for the ML application at hand, the optimal quantization configuration in terms of both model size and energy consumption reduction can be found automatically with AutoQKeras.

A block diagram representation related to the complete workflow for automatically converting any uncompressed Keras model into its optimal heterogeneously quantized-aware training version through the AutoQKeras optimization, is given in Figure 3.13.

Specifically, starting from the Keras model and knowing the accuracy requirements and resource constrains of the DNN model at hand, after obtaining the corresponding QKeras quantized version and estimating its size and energy consumption with QTools, it is possible to automatically find the optimal quantization configuration for the specific case through the use of AutoQKeras.

As a result, the automatic training-aware procedure of heterogeneous quantization is a step ahead of standard manual post-training quantization. This method of automatically finding the optimal heterogeneous configuration of quantized parameters for a QKeras model starting from the corresponding Keras version is of significant interest for low-power and resource-constrained embedded applications. In FPGA-based DNN inference accelerators, where the amount of resources is limited, it allows to choose the optimal bit width automatically to improve DNN processing speed and energy efficiency, while keeping the model accuracy similar to the reference fp32 DNN model, but with a substantial reduction in model size and energy consumption.

As an example, Figure 3.14 (on the left) shows a comparison of the normalized accuracy of

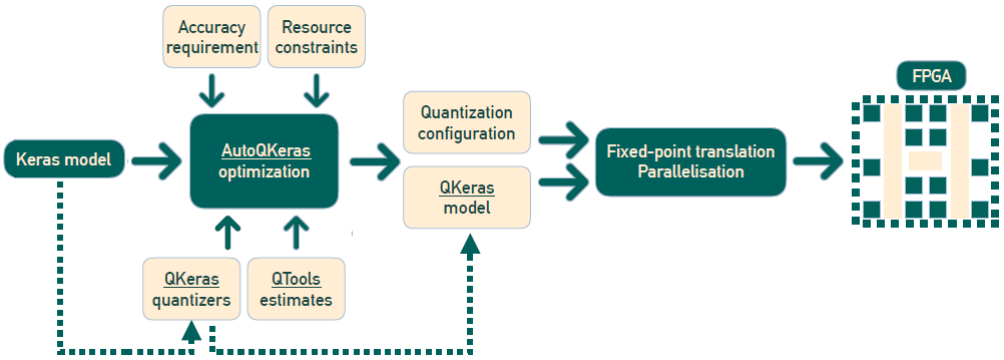

Figure 3.13: Workflow for converting an uncompressed Keras model into the optimal heterogeneously quantized version for FPGA implementation through AutoQKeras. Source [49]

a homogeneously post-training and QKeras quantized model achieved through the AutoQKeras optimization, as a function of bit width, with respect to the reference model with fp32 precision. While post-training quantization suffers a rapid degradation in model accuracy starting at bit widths below 14, the QKeras model quantized through quantization-aware training maintains approximately the same accuracy as the reference fp32 model up to a bit width of 6 for weights and activations.

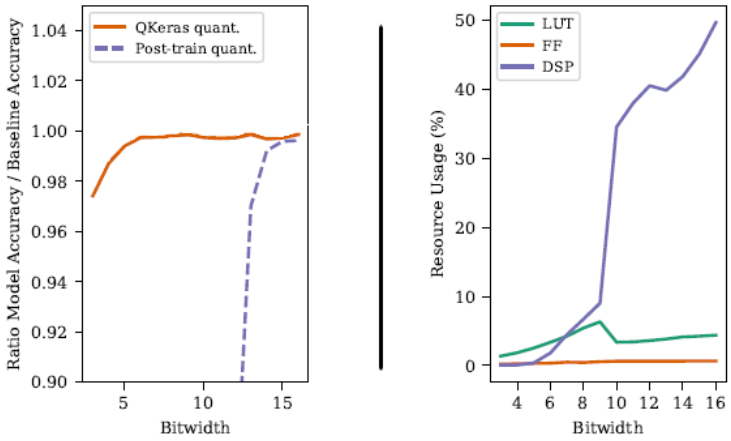

Figure 3.14: Post-training quantized and QKeras quantized model accuracy in the case of homogeneous quantizations normalized with respect to reference model with fp32 precision as a function of bit width (on the left) and FPGA resource utilization percentage of a quantized QKeras model as a function of bit width (on the right). Source [49]

In the Figure 3.14 (on the right) is reported the percentage of FPGA resource utilization for a homogeneously QKeras quantized model through AutoQKeras optimization as a function of the bit width. There is a clear reduction in all types of FPGA hardware resource consumption for implementing the same DNN model as the bit width used for the representation of quantized data decreases, except for LUTs, that show an increasing for a certain bit width value, and then return to decrease. The most significant reduction is in the percentage of DSPs used to perform the MACs required for DNN inference processing, due to the fact that if the precision chosen for the representation of parameters and activations falls below that of the DSPs, then the execution of the

MACs is moved to the LUTs, because it is more convenient. Thus, it also explains the increment in the use of LUTs in correspondence where the decrease of the curve relative to the DPS is maximum.

To conclude, the heterogeneous quantization-aware training method automatized with AutoQKeras is very attractive for FPGA-based DNN inference accelerators, because since the number of DSPs available within the chosen processing unit is usually the limiting factor for the implementation of DNN models, moving the execution of MACs to LUTs allows the implementation on the same amount of hardware resources of more complex DL algorithms, allowing to increase the achievable accuracy for the task under consideration.

3.6.2 Model Compression to Reduce the Number of Operations and Model Size

The other class of hardware-oriented optimization methods for approximate computation is those that focus on reducing the number of operations (i.e. MACs) and model size through so-called *model compression*. More in detail, these approximation methods are based on the assumption that the larger the size of the DNN model, the higher the level of achievable accuracy and the amount of hardware resource consumption in terms of both computational and memory requirements. Consequently, reducing its size in terms of number of MACs per DNN inference iteration, and thus the number of memory accesses to fetch the weights and activations to be processed, means trading a decrease in accuracy for an increase in speed and energy efficiency, in addition to a saving in hardware resource consumption.

– *Network pruning* [50] is a model compression technique that removes the weights and the neurons that do not influence the accuracy significantly, either eliminating the correspondent connections or directly the associated unit. More in detail, as state in [51], in order to optimize the training in order to achieve a better inference accuracy, DNN models usually include a much larger number of weights than the minimum required, resulting to be *over-parametrized*. Thus, through network pruning, it is possible to remove redundant neurons and weights of a trained DNN model, accelerating the execution of the inference. It is done by eliminating this redundancy from the trained DNN model setting to zero the weights not too significant. Not to influence to much system accuracy, the pruned DNN model is fine-tuned in order to balance the accuracy loss by modifying the value of the remaining connections. Thus, network pruning makes sparser the DNN model in terms of weights on which it is applied.

  More in detail, it is possible to prune any DNN model in two different ways (see Figure 3.15).

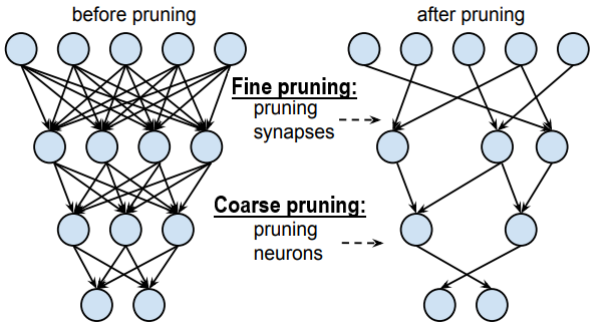

Figure 3.15: Comparison of the internal structure of a Neural Network before pruning (on the left) and after fine and coarse pruning (on the right) methods . Source [52]

1. The *fine pruning* (also referred to as *weight pruning*) consists in setting to 0 only the low value non-significant connections between nodes contained in consecutive layers of the DNN model, and then eventually fine-tunes the pruned DNN model to recover some accuracy loss. Fine-tuning or not fine-tuning the remaining weights after pruning determines the level of aggressiveness with which the fine pruning can be applied. Thus, it results in a significant change in the amount of weights of the original DNN model that can be removed, make it sparser. As an example, the results reported in [53] show that in AlexNet, the removal of weights with a low value through the fine pruning and the subsequent fine-tuning of the remaining ones allows to reduce the number of CONV and FC layers weights by 53 % compared to that of the uncompressed case, for less than 0.5 % accuracy loss. More in detail, in terms of reduction in the number of weights of the DNN model obtained through the fine pruning application, the largest contribution comes from reducing their number in the FC layers, being their amount in terms of DNN model parameters the dominant one compared to that of the CONV layers, as shown in Figure 1.15. As a result, the reduction in the number of weights achieved compressing the model by fine pruning the DNN model results in a fewer number of MACs required to perform each DNN inference iteration, since when the weight has null value, the result of the multiplication is already known. Thus, it results also in a less number of memory accesses, making it an attractive technique for FPGA-based DNN inference accelerators to increase throughput/latency and energy efficiency. Then, there are other methods to decide whether a weight can be pruned or not than just thinking its magnitude proportional to the loss in accuracy. If the goal is to decrease the energy consumption by fine pruning some DNN model parameters, making an energy efficiency estimation based only on the amount of weights in each DNN model layer is not very truthful, as there are other factors to consider. In fact, not necessarily the DNNs with a larger amount of weights consume more than DNNs with less parameters, but all types of data involved in the DNN inference should be considered to obtain a more reliable energy estimation, as shown in Figure 3.16 for the GoogLeNet case.

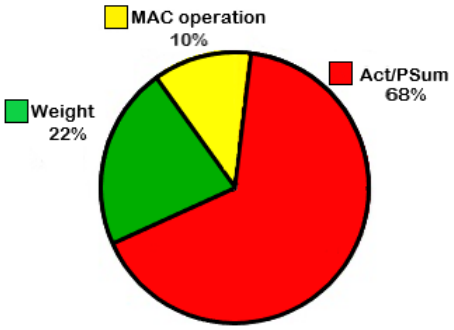

Figure 3.16: Energy consumption contributions for GoogLeNet inference processing. Source [54]

Referring to the Figure 1.15, although the number of parameters in the FC layers in much larger than that in the CONV layers, the energy consumption of the latter is significantly higher than that of the former, due to the much larger number of MACs to be performed, each one involving four memory accesses. In addition, since the trend is to have deeper and deeper DNN models with an increasing number of CONV layers, an alternative criterion to base the application of fine pruning with respect to the magnitude-based one need to be considered in order to improve energy efficiency. Thus, instead of basing the fine pruning according to the magnitude of the

weights, an alternative is to base the removal of weights on reducing the overall energy consumption of the DNN model required both for moving the data to be processed and for performing the MACs. More in detail, it requires to make an energy estimation of the DNN model layers in order to sort them basing on their energy cost and then pruning the connections of those that consume the most energy. Figure 3.17 shows a comparison between top-5 accuracy versus energy consumption for standard DNN models, focusing on the impact of the two different types of fine pruning.

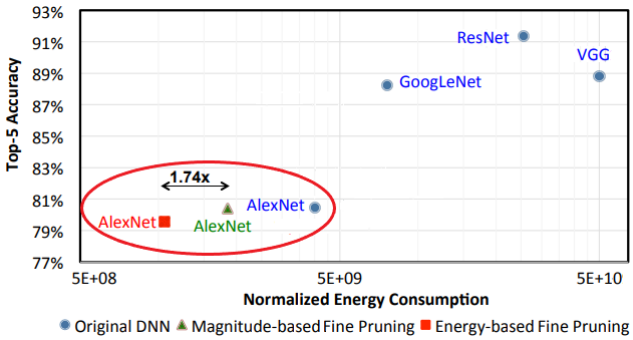

Figure 3.17: Comparison of top-5 accuracy versus energy consumption for standard CNNs, with a focus on AlexNet. Source [55]

More specifically, the energy-based approach turns out to be more energy efficient than the magnitude-based one in the case of AlexNet examined in [53], (see Figure 3.18) with approximately the same value of accuracy.

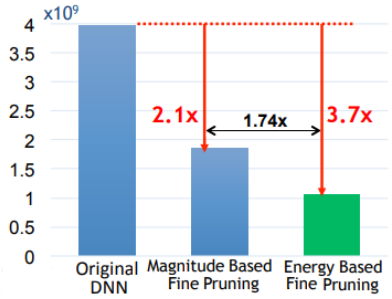

Figure 3.18: Comparison of energy savings obtained by applying the magnitude-based and energy-based fine pruning methods in AlexNet. Source [55]

2. The *neuron pruning* (also referred to as *coarse pruning*) consists to directly eliminate the least important whole neurons. More in detail, reducing the amount of weights in the DNN model not only results in an improvement in speed and energy consumption, but also not results in an increasing demand of additional hardware resource for sparsity handling for FPGA- DNN inference accelerators, which is a very important aspect in the case of embedded systems. Although the fine pruning allows to a strongly network compression, it results in an irregular sparsity pattern for weights of the DNN model, that may require an elevated amount of additional hardware to be handled for accelerating DNN inference processing. Coarse pruning allows to obtain an even more efficient DNN inference processing introducing a more regular weight distribution in

the DNN model. A flow-chart related to the network pruning technique performed iteratively with a high level of granularity regarding the removal of weights until the required compression factor or the maximum acceptable level of accuracy loss for the application under consideration is reached, is shown in Figure 3.19.

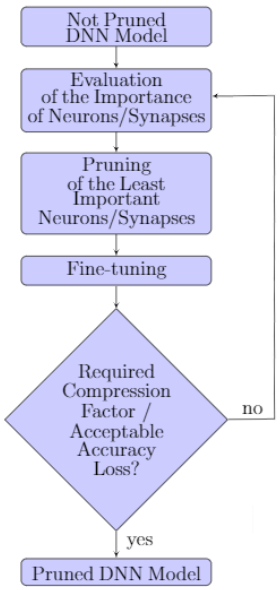

Figure 3.19: Flow-chart for performing fine/coarse pruning technique iteratively.

In addition to speed and energy efficiency improvement, sparsity introduced by the application of both network pruning types in DNN models also allows for advantages at the level of storing weights in memory. More in detail, it is possible to represent the sparse weights in a data format that, depending on the *sparsity compression method* used, to reduce the memory size and improves the memory bandwidth.

– The *statistics* exhibited by the *activation function* can be seen as a kind of model compression technique taking advantage of the non-linearity layers to increase the level of sparsity of activations in the DNN model. As anticipated in Section 1.2, ReLU is the most advantageous choice in DNN models for this purpose, since outputs a zero value for all negative activation values it receives as input (see Figure 3.20). In addition, the hardware resource consumption cost for its implementation is extremely low, since to determine the output it is sufficient to make a comparison among the considered input and the zero value, making it the optimal choice for FPGA-based DNN inference accelerators implemented on resource-constrained systems.

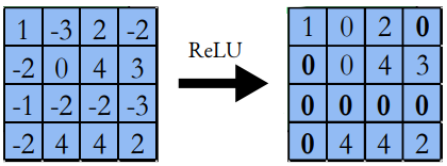

Figure 3.20: Example of sparsity increment in activations after ReLU application. Source [56]

Sparse activations due to the application of ReLU can be compressed before being stored through some *sparsity compression methods*, in order to both reduce the memory size and the memory bandwidth requirements, significantly increasing the performance of the FPGA-based DNN inference accelerator. Figure 3.21 shows a comparison between the number of total activations and the number of non-zero activations along the CONV layers of AlexNet after applying ReLU. It results in a effectively sparse DNN model. More in detail, it is evident that the deeper it goes into the DNN, the more the percentage of non-zero activations decreases, due to multiple applications of ReLU after each CONV layer.

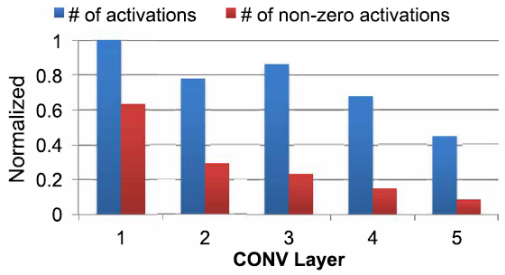

Figure 3.21: Activation distribution before (in blue) and after (in red) ReLU in AlexNet. Source [6]

– *Knowledge distillation* is a model compression method for training a smaller DNN model from the results of a larger one, in order to achieve the same level of accuracy. While increasing the number of layers and neurons of a DNN model is the standard way to achieve higher accuracy for a given task, at the same time it requires higher computational and memory requirements, resulting to be a problem for systems with a limited number of resources, such as embedded devices for edge computing. To maintain a low DNN model size with an high level of accuracy, and thus have an high speed and energy efficiency for DNN inference processing with a low hardware resource consumption, it is possible to transfer the learning (i.e. the values of the weights of a trained DNN model required obtain a certain level of accuracy in the results for a given task) to solve a given problem, from a complex (referred to as *teacher*) to a simpler DNN model (referred to as *student*) through so-called knowledge distillation. More in detail, the idea is to train the teacher with a certain set of input data, and then to transfer the acquired knowledge to the student training the latter with the outputs produced by the former, used either directly as training data [57], or as hints to couple with the original set of input [58].Thus, the complex teacher acts as a kind of intermediate step used to extract the most important features from the original set of training data for performing the task at hand, and then they are used as training data for the simpler student. This knowledge transfer allows the student to achieve a level of accuracy that would not be achievable if it was directly trained with the same set of data of the teachers. Thus, the student imitates the solution found by the teacher with comparable performance in terms of accuracy, but with an higher speed and energy efficiency for network compression. In some cases, the accuracy of the student can be higher than teacher one, since a reduction in the complexity of the DNN model implies a lower risk of overfitting. The generic structure of a knowledge distillation system is shown in Figure 3.22.

  As a result, replacing a complex DNN model with a shallower one through knowledge distillation allows accelerating DNN inference and reducing hardware resource consumption, while maintaining the same level of accuracy. The high architectural flexibility exhibited by FPGAs in designing of custom logical units makes this technique of considerable interest in order to improve the speed and energy efficiency of FPGA-based DNN inference accelerators by combining it with additional approximation techniques.

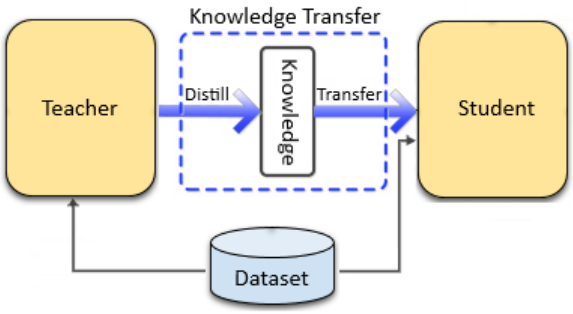

Figure 3.22: Representation of the knowledge distillation model compression technique. Source [59]

– *Structural simplifications* include a set of model compression methods that allow to move towards smaller DNN models in terms of both the number of parameters and MACs required to perform a DNN inference iteration, without sacrificing too much accuracy. While the usual approach in order to improve the DNN model accuracy is to increase the number of DNN layers (especially the number of CONV layers), these methods allow to implement DNN models with a reduced complexity in terms of parameters and MAC operation compared to standard DNN models with approximately the same level of achievable accuracy through some architectural choices. More in detail, the aim is to decompose large convolution kernels into a series of smaller ones that, applied sequentially, allow to obtain the same receptive field, but with a fewer number of weights. More in detail, it is possible to apply this idea either pre-training the DNN model, by building the network with a series of small convolution filters instead of with only large one, or post-training the DNN model, by decomposing the already defined convolution filters through a low-rank factorization method called *Tensor Decomposition*, and then fine-tuning the resulting kernel weights to recover some accuracy.

1. Regarding the *pre-training approach*, it is possible to replace large convolution kernels with a series of smaller ones, which when applied on the input future map multiple times sequentially allow to obtain the same elements for the output feature map, with a lowering the number of required kernel weights (see Figure 3.23). As an example, it is possible to obtain the same output feature map by performing the convolution operation among the starting input feature map and one 5×5 convolution kernel, or by decomposing it into a series of two 3×3 convolution kernels (see Figure 3.23). It results in a reduction in the number of kernel weights from 25 to 18 to obtain the same result, with a substantial decreasing in the DNN model computational and memory requirements due to the reduction of weights , at the expense of a certain loss of accuracy when the obtained receptive field is slightly different than the not decomposed one.

   The most common DNN models with compact network architectures based on the CONV filter decomposition before the training able to reduce the number of weights while maintaining high accuracy levels are the so-called *SqueezeNet* [60] and *MobileNet* [61]. Both of these compressed DNN architectures are very interesting DL algorithm hardware implementation in embedded devices for edge DNN inference processing due to their small hardware resource consumption combined with both an high speed and energy efficiency.

   (a) *SqueezeNet*. Since the number of kernel weights of a convolution filter depends on both its height and width and its number of channels, in SqueezeNet the reduction of network parameters is obtained through the use of 1×1 convolution filters, able

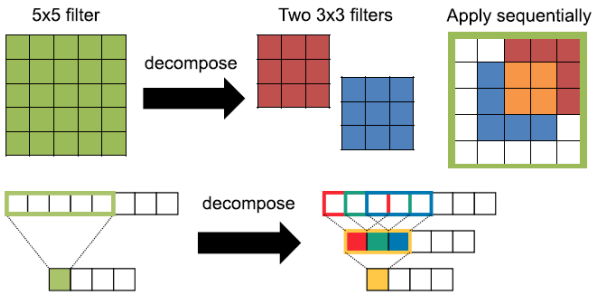

Figure 3.23: Decomposition of a 5×5 convolution filter into two successive 3×3 convolution filters to reduce the number of weights, while maintaining the same receptive field. Source [54]

to reduce both the number of weights in the current layer filter and the number of channels in the next filter layer. SqueezeNet is derived from the AlexNet architecture, and allows to achieve the same accuracy, but with a a reduction in the number of network parameters by about a factor 50×. More in detail, to reduce the filter size of the current CONV layer most of the 3×3 kernels of AlexNet are decomposed into a series of 1×1 kernels applied sequentially, reducing the number of kernel weights by a factor 9×). It allows to reduce also the number of channels of the next CONV layer, since by applying to the input feature map 1×1 convolution filters results in an output feature map with the same number of channels as the number of convolutional filters applied. Thus, for obtaining this reduction it is necessary that for 1×1 convolution filters the number of filters is less than the number of channels in each one, as shown in [62]. In addition, since SqueezeNet model is small, the POOL layer operation is postponed as late as possible within the network architecture, so that the the feature maps for the layers in between are large. Thus, the deeper the dimension of the feature map is reduced, the more information is retained in the intermediate layers of the DNN model, thus increasing the accuracy of the outputs produced. Specifically, in SqueezeNet the reduction of weights is realized through the so-called *fire module* (see Figure 3.24 on the right), composed of two different layers, which are the *squeeze layer* and the *expansion layer*. While the former consists of a set of 1×1 convolution filters dedicated to reduce the number of channels of the input feature map of the current layer into one for input feature map of the next layer, the latter includes a sequence of both 1×1 convolution kernels to reduce the number of channels in the output feature map starting from its incoming input feature map, and 3×3 convolution kernels, which are used to extract features from the data to be processed at a higher level of detail than the one possible through 1×1 convolution kernels. Thus, through the combination of these two convolution filters with different sizes it is possible to increase the level of accuracy, and at the same time reduce the number of kernel weights. As shown in Figure 3.24 (on the left), the architecture of SqueezeNet is composed of 8 fire modules able to make convolution operation with a reduced number of kernel weights, and by only two CONV layers as first and last layer, allowinf to reduce greatly the DNN model size. In particular, in SqueezeNet there are any FC layers, which are the most expensive one in terms of number of parameters, and so the most memory expensive (see Figure 1.15). Thus, the absence of FC layers and the CONV layers implemented as fire modules allow to reduce greatly the model size compared to standard DNN models, while maintaining an high level of accuracy. However, while reducing the number of weights allow to reduce the number of MACs

operations and memory requirements for DNN inference iteration, it is not always associated with a reduction in energy consumption per DNN inference iteration.

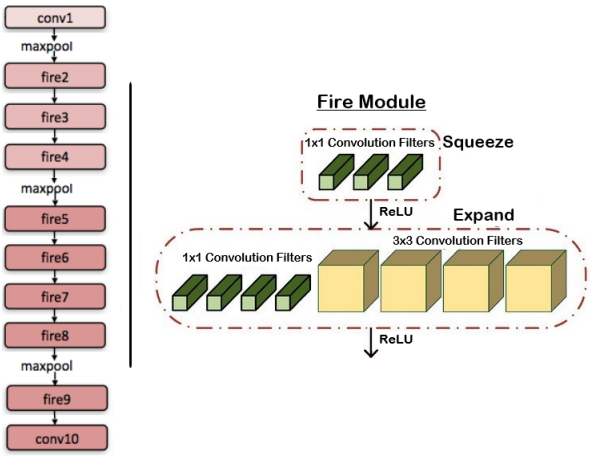

Figure 3.24: The internal architecture of SqueezeNet (on the left) consists of a sequence of fire modules (on the right) and a single CONV layer as input/output layer. Source [63]

In Figure 3.25 it is possible to see that SqueezeNet consumes a higher amount of energy than AlexNet for the same task, with approximately the same level of accuracy, despite the fact that the number of weights of the former is much lower than that of the latter.

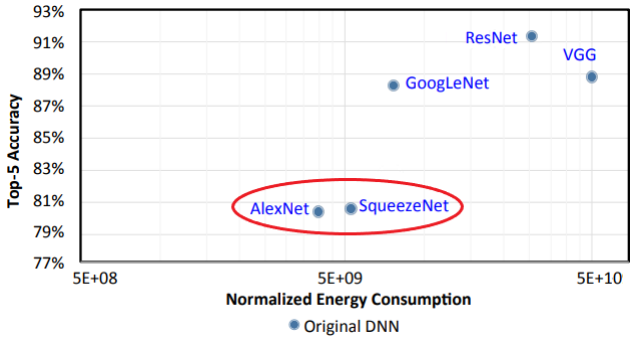

Figure 3.25: Comparison of top-5 accuracy versus normalized energy consumption for CNNs, with a focus on AlexNet and SqueezeNet. Source [55]

(b) *MobileNet*. Reducing in the number of kernel weights in MobileNet is accomplished with an alternative approach through which to perform the convolution operation than the one standard CNNs. More in detail, instead of performing it sliding the convolution kernel on the input feature map for all its channels, through the so-called *depth-wise separable convolution operation* it is possible to perform it by decomposing each CONV layer into a sequence of two different CONV layers, which are a *depth-wise convolution* followed by a *point-wise convolution* (see Figure 3.26). While the former decomposes each standard convolution filter in a number of single-channel *depth-wise convolution filters* equal to the channels of the standard convolution filter itself and with the same dimension in terms of height and width to

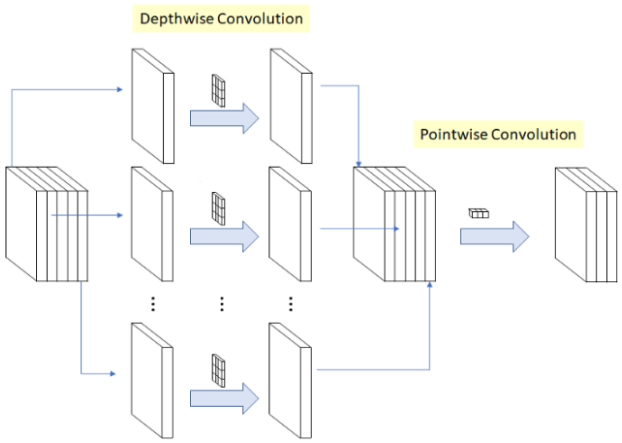

Figure 3.26: Graphical representation of the depth-wise separable convolution operation in MobileNet. The standard convolution operation composed of N convolution filters is performed first dividing each M channels convolution filter into a sequence of M depth-wise convolution kernels to be applied separately to each input feature map channel, then combining the achieved output feature map channels with N point-wise convolution filters. Source [64]

be applied separately to each channel of the input feature map, the latter combines the channels of the achieved output feature maps through the application of an amount of 1×1 *point-wise convolution filters* equal to the number of standard convolution filters (see Figure 3.27).

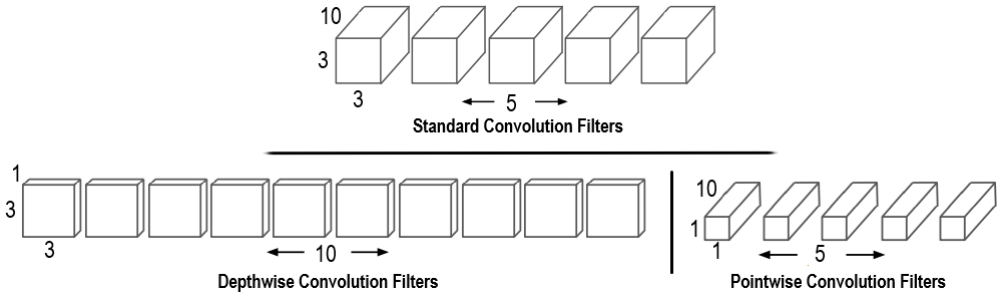

Figure 3.27: In MobileNet the standard convolution filters (on the top) are replaced by sequences depth-wise convolution filters equal to its number of filters and long as its number of channels with the same height and width (on the bottom left) plus a sequence of 1×1 point-wise convolution filters (on the bottom right) long as its number of filters, to build a depth-wise separable filter.

As an example [65], assuming to have a CONV layer composed of 5 3×3 convolution filters, each one consisting of 10 channels. While if the convolution operation is performed as in the standard CONV layers, it would require $5 \times 3 \times 3 \times 10 = 450$ weights, if it is performed as a depth-wise separable convolution operation it requires only $10 \times 3 \times 3 \times 1 + 5 \times 1 \times 1 \times 10 = 140$ weights, which in turn results in a decrease in both the number of MACs per DNN inference iteration and memory requirements,and thus in both speed and power consumption improvements. Figure 3.27 shows a comparison for the considered example of the convolution kernel

organization in the cases of standard and depth-wise separable convolution operation. The internal architecture of MobileNet consists of a succession of CONV layers implemented through the use of only depth-wise separable convolution layers, with the exception of the first CONV layer, which is based on the standard convolution operation. Unlike SqueezeNet, the output of the last CONV layer is followed by a POOL layer, and then by an additional FC layer. Thanks to the pooling operation preceding the last FC layer, it does not imply in a large increase in the number of DNN model parameters. Thus, performing the convolution operation in this way leads to a substantial reduction in the number weights for each CONV layer, and thus of memory requirements, resulting in a substantial increase in performance and energy efficiency of the of the system, with a significantly reduced model size.

2. Regarding the *post-training approach*, in order to reduce the number of weights of CONV and FC layers in an already trained network, and so to improve the DNN inference processing both in speed and energy efficiency, it is possible to break both of them into a succession of separable smaller size filters using some *low rank approximation methods*, with a reduced impact on the final accuracy value of the system, that anyway can be recovered thorough fine-tuning the DNN model weights. Unlike to having a small number of large layers, increasing the number of smaller layers in the network architecture results in an increase in the compression rate of the DNN model in terms of the amount of weights, and thus an increase in performance, at the expense of some loss of accuracy. More in detail, while FC filters are 2-D kernels involving a matrix-vector operation with the activations, the dimensions of CONV filters are 4. For CONV layers it is possible to divide through some decomposition methods, such as the *tensor decomposition* one, large 4-D kernels into a series of two smaller 3-D kernels, in order to reduce the number of MACs required per convolution operation, thus resulting in a computational and memory demand reduction per DNN inference processing. For FC layers, it is possible to reduce the number of weights following the same idea through the so-called *singular value decomposition*, where the associated weight matrix is divided in the multiplication of two smaller weight matrices, obtaining the same improvement for DNN inference processing.

As a result, such architectural simplifications that enable the development of simpler DNN models are ideal for FPGA-based DNN inference accelerators, allowing for both speed and energy efficiency improvements compared to the case of orginal complex DNN model implementation, at the expense of some loss of accuracy, that can be restored through a fine-tuning operation of the weights.

3.6.3 Data Compression Methods to Exploit Weight and Activations Sparsity

As mentioned above, through the application of network pruning methods and the exploitation of the ReLU statistics it is possible to make sparser the DNN model, at the expense of a certain loss in accuracy. More in detail, these model compression methods allow to reduce the number of MACs involving a multiplication between at least one null weight/activation and the corresponding memory accesses. It results in a better use of the on-chip memory, reducing the number of memory accesses to the off-chip one, involving in an acceleration and energy consumption saving for the corresponding FPGA-based DNN inference accelerators on which the sparser DNN model has been implemented. Thus, avoiding the execution of MACs with a null operands reduce both computational (in terms of number of PEs) and memory requirements (in terms of memory bandwidth and size) optimizing the speed and energy efficiency of the FPGA-based DNN inference accelerator.

Thus, to maximize the exploitation of approximating methods able to make sparser the DNN model in terms of memory, it is possible to adopt the so-called *sparsity compression methods* to compress the sparse weights and activations before saving them in memory.More in detail, these compression techniques allow to reduce the cost in terms of memory requirements and data movement, it is possible to avoid the saving of zero activations and weights by storing only the non-null network parameters and their indexes in the corresponding matrix (see Figure 3.28).

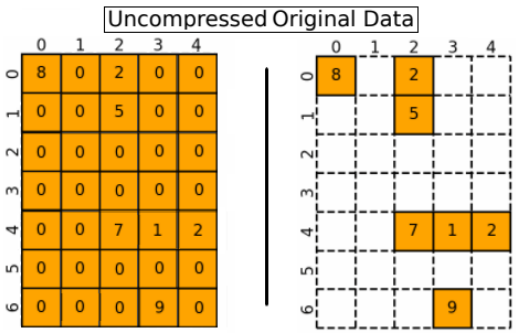

Figure 3.28: Dense matrix (on the left) versus sparse matrix (on the right). Source [66]

This allows to obtain a high compression factor for the network parameters to be saved in memory necessary for the processing of a DNN inference iteration. In this way, the number of external DRAM accesses is greatly reduced, if not completely zeroed out, since by doing so many times the size of the internal SRAM is sufficient to store all the parameters required for processing a DNN inference iteration.This is of interest with respect to FPGA-based DNN inference accelerator, where the amount of on-chip memory is limited and the off-chip memory accesses determine the main limiting factor both in terms of speed and energy efficiency, since the compression of the stored data allows to maximize the reuse of the data at the low levels of the memory hierarchy required for the MACs per DNN inference iteration.

More in detail, it is possible to compress sparse weight and activation matrices with respect to both rows and columns, denoted as *Compressed Sparse Row* (CSR) and *Compressed Sparse Column* (CSC), respectively (see Figure 3.29).

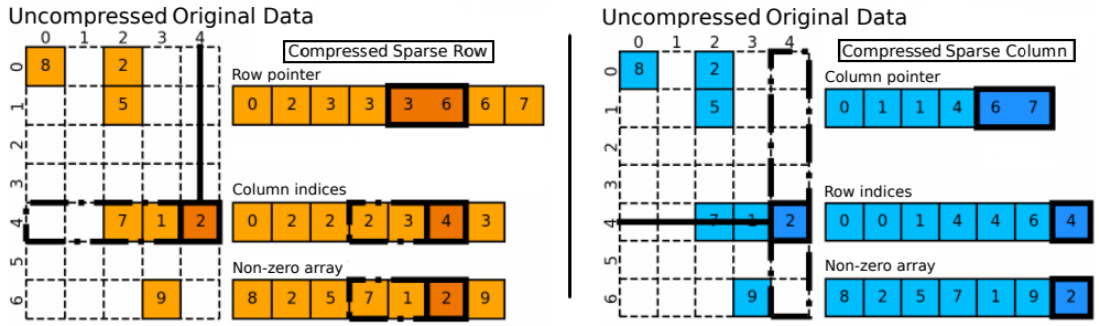

Figure 3.29: Graphical representation of Compressed Sparse Row (on the left) and Compressed Sparse Column (on the right) data compression methods.

These techniques are able to compress efficiently a sparse matrix through the use of two types of data and three different arrays, which are one data array containing all the non-null values of the original matrix in row/column order for CSR and CSC respectively, called a *non-zero array*,

and two index arrays needed to return to the compressed matrix to the uncompressed original one. While the former, referred to as the *column/row index array* respectively for CSR/CSC, contains the row/column indices corresponding to the non-null elements of the uncompressed original matrix, the latter, referred to as the *row/column pointer array* respectively for CSR/CSC, consists of a set of row/column pointers that incrementally maintains the count of the number of non-null elements per row/column of the uncompressed original matrix. Specifically, in the row/column pointer array, pairs of adjacent elements mean two things. While the position refers the number of row/column (element #0 and element #1 indicate the first one, element #1 and element #2 indicate the second one, and so on), the difference between their values represents the number of non-zero elements in the row/column to which such a pair refers. As a result, through the CSR and CSC sparsity compression methods, it is possible to compress a sparse matrix into a set of vectors to be saved in memory, which can be accessed by rows or columns respectively, thus reducing storage requirements.

In the two above mentioned sparsity compression techniques it is necessary to customize the DNN model architecture by implementing dedicated logic units either to perform processing directly with the compressed network parameters, or for the decompression of weights and activations stored in memory. Thus, these compression method are of particular interest for the handling sparse matrices to store in memory for FPGA-based DNN inference accelerators, due to the high architectural flexibility of FPGAs.

An alternative sparsity compression method widely used for optimizing the storage of of sparse weight and activation matrices is the so-called *Compressed Image Size* (CIS) (see Figure 3.30). More in detail, it reduces the hardware consumption for the decompression of the stored sparse data, since it is not required.

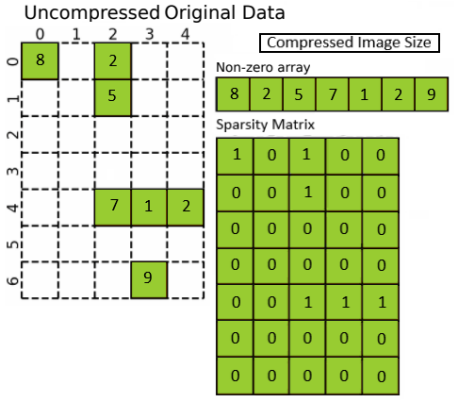

Figure 3.30: Graphical representation of Compressed Image Size sparsity compression method.

More in detail, it is composed only of two types of data and two different arrays. While the *non-zero array* is an array of non-null values contained in the uncompressed matrix in row order, the *sparsity matrix* is a matrix of the same size as the original one, which contains only 0 and 1 values, used to take into account the position of the elements contained in the non-zero array. Comparing the various sparsity compression methods, it is evident that through the CIS method it is possible to obtain a good compression ration for sparse network parameter at the minimum possible overhead in terms of resource consumption to perform data compression. Thus, it results of particular interest for resource-constrained devices, such as FPGA-based DNN inference accelerators for edge computing.

To summarize, while exploiting parameter sparsity introduced by model compression methods allows to accelerate the DNN inference processing on FPGA-based DNN inference accelerators

by reducing the amount of MACs, the reduction of the number of memory accesses required to perform an inference iteration is achieved with the introduction of the sparsity compression methods. It results in energy savings related to the avoidance of execution of the MACs involving a zero operand and the reduction in the off-chip memory accesses, at the expense of the design of custom hardware to decompress the stored values or to work directly on compressed network parameters. While fine pruning removing individual weight introduces an high level of irregularity in the data pattern, resulting in complex memory accesses, coarse grain pruning removing directly a neuron determine a more regular data pattern. It makes the compression operation simpler, resulting in a lower memory requirement for the storing of sparse network parameters. Thus, for FPGA-based DNN inference accelerators, due to the flexibility of such hardware platforms that makes it possible to implement custom logic units, the exploitation of sparsity compression methods represents an opportunity in order to achieve substantial improvements in the execution of the DNN inference process at the edge.

Chapter 4

Conclusions and Perspectives

4.1 Conclusions

In last years, DNNs have become the most widely used computational model in most ML applications, due to the high level of accuracy they are capable of achieving. As a consequence, the need for ever deeper ANNs has significantly increased the algorithmic complexity, and consequently the computational power and storage requirements in terms of both amount of operations and memory transfers for DL algorithm processing.

As a result, the development of dedicated *hardware accelerators* capable of exploiting the different types of parallelism exhibited by DNN models becomes of paramount importance to improve the performance in DNN processing both in terms of speed and energy efficiency. While the training remains primarily devolved to cloud-based execution to maximize computational performance, the current trend is to move the inference processing at the edge on resource-constrained embedded devices, to avoid the increase in latency and power consumption to transfer the DNN model and data to be evaluated to the cloud. Thus, the choice of the hardware platform that best suits DNN inference processing for low-latency/low-power embedded ML applications becomes of paramount importance.

In this scenario, custom hardware processing units, have emerged as the best compromise for accelerating DNN inference at the edge due to the high throughput/latency and low power consumption they can achieve compared to GPPs. In addition, due to their small size in terms of space occupancy, they can be easily implemented on embedded devices with a limited amount of hardware resources. Due to the high cost in terms of design, the low level of architectural flexibility, and the total lack of reprogrammability, ASIC-based solutions are scarcely used, preferring FPGA-based ones. Although the lower operating frequency of FPGAs compared to alternatives results in reduced system power consumption, at least initially it can be a limiting factor in terms of maximum achievable throughput/latency.

However, the lower frequency of FPGAs compared to alternatives is not a problem, because the high level of parallel execution, the elevated architectural flexibility in terms of specific hardware logic unit design and the possibility of being re-configurable multiple times allow FPGAs to achieve high performance in terms of speed and energy efficiency by maximizing the exploitation of the inherent parallelism exhibited by DNN models through the development of hardware architectures with efficient data flows able to maximize the reuse of the data, and the application of hardware/software-level optimization methods.

Table 4.1 summarizes the main advantages and disadvantages of using FPGAs as processing units to accelerate DNN inference processing on embedded devices. Specifically, if the requirements of the ML application under consideration allow a minimal loss of precision on the results

Table 4.1: Advantages and disadvantages of using FPGAs to accelerate DNN inference processing on embedded devices compared to CPUs, GPUs and ASICs.

FPGA advantages	**FPGA disadvantages**
High level of parallel execution	Low working frequency
Elevated architectural flexibility and ability to design custom hardware	Reduced availability of hardware resources
High throughput and low latency	Complex design flow
High energy efficiency	Relatively long reconfiguration time
Ability to be reconfigured multiple times	Low flexibility in performing different tasks for the same architectural configuration
Small area occupancy	
Deterministic scheduling can support hard real-time applications	

produced by the corresponding DNN model, it is possible through the application of a wide range of approximation methods to increase performance in both speed and energy efficiency. Making a comparison through a *roofline model* analysis in terms of theoretical maximum performance obtainable from different hardware platforms available for the processing of DNN inference, which then takes into account the amount of PEs available to the hardware platform in question and the bandwidth of external memory with which it interfaces, FPGAs are found to be the most promising solution in terms of performance when it is exploited in the approximation of computation in terms of both reduction in the number and precision of arithmetic operations that compression of the size of the model.

Hence, these improvements in both speed and energy efficiency for FPGA-based DNN inference accelerators are achieved through a design that consider optimizing the most critical parts of the system that limit throughput/latency and contribute most to energy consumption. As a result, identifying the key issues is critical for understanding how to best balance the various system design metrics through the application of a wide range of techniques. Specifically, they are based on one side on the development of hardware architectures with highly parallel computing paradigms with energy-efficient data flow, and on the other side on the implementation of optimization algorithms with the goal of improving the speed and energy efficiency of the system, possibly scarifying the smallest possible amount of accuracy. These optimizations are developed in order to make FPGA-based DNN inference acceleration a feasible solution for the ML application under consideration.

This thesis work consisted of a detailed review of the state of the art in FPGA-based DNN inference accelerator designs, focusing primarily on analyzing the improvements that can be achieved through both efficient architectural design and trough the application of hardware/software optimization algorithms in order to accelerate DL inference on custom hardware processing units.

More in detail, for speed the focus is on maximizing execution parallelism within the limited available PEs for MAC operation implementations and the bandwidth requirements of the off-chip memory containing the DNN model to be evaluated. For energy efficiency, the goal is to optimize all aspects of data movements and external memory accesses, as this is the bottleneck for DNN processing. In this scenario, the FPGA spatial architectures on which FPGAs are based allow to reduce the energy consumption impact of the DNN model due to the interaction of the system with the off-chip memory by distributing part of the storage elements directly into PEs through the implantation of an on-chip memory hierarchy. The introduction of such an architectural improvement, combined with the development of energy-efficient data flows, maximizes the

reuse of data stored in on-chip memory, avoiding to reduce the accesses to off-chip memory. Unlike improvements at the architectural design level that do not affect model accuracy, hardware/-software optimization methods to accelerate DNN inference processing usually imply some loss of precision in the produced result compared to that of the original DNN model.

By exploiting the over-parametrization condition of DNNs useful for facilitating their training, the application of *hardware-oriented optimization algorithms for approximate computation* is particular attractive for accelerating DNN inference processing on FPGAs, while also allowing for reduced hardware resource usage. Experimental results have shown that this class of optimization algorithms is the most effective for improving the performance of DL implementations on reprogrammable custom hardware processing units.

In more detail, if the ML application at hand accepts a moderate loss in the precision of the result produced, the implementation of these approximating methods on FPGA-based DNN inference accelerators saves computational resources and improves speed and energy efficiency significantly. This is achieved either by reducing the precision of computation through *data quantization*, thus moving from single-precision floating point to lower bit width fixed point data format for representing weights and activations, or by reducing the number of MAC operations and model size by increasing the level of sparsity of network parameters to be processed by *model compression*.

Relative to the reduction in computational precision, the number of bits on which to quantize is limited by the system accuracy requirement for the ML application under consideration. Specifically, experimental results have shown that heterogeneous quantizations for weights and activations on a variable number of bits outperform homogeneous ones.The flexibility of FPGAs supports the management of variable bit widths for representing both different types of network parameters within the same layer and for those of weights and activations of different layers, allowing for additional performance. Then, it has been described a methodology to automatically find the optimal heterogeneous quantized configuration for a given DNN model to achieve a balance between reducing resource consumption, latency and power consumption while, maintaining high accuracy.

On the other hand, increasing the sparsity level not only allows to reduce the number of MACs per inference iteration and the size of the DNN model by eliminating the execution of computations where at least one of the two operands to be multiplied is null, but also to better manage the amount of storage elements available through the development of *sparsity compression methods*, capable of optimizing the storage of sparse parameters. The architectural flexibility exhibited by FPGAs makes it possible to exploit the sparsity obtained through the compression of the DNN model to speed up the execution of the edge inference process.

In this dissertation, several methods for approximating computation have been analyzed and compared in order to highlight the positive and negative effects of their application with respect to the hardware implementation of complex DNN models on FPGAs.

As a result, trading-off a certain amount of model accuracy, it is possible to achieve significant improvements in both speed and energy efficiency by increasing parallelism and reducing the number of operations and memory accesses. Furthermore, due to the architectural flexibility of FPGAs, it is possible to apply a combination of these techniques to obtain better and better results. Hybrid strategies allow the joint use of multiple computational approximation techniques within the same FPGA-based DNN inference accelerator design in order to achieve better performance than in the case of applying a single method. As a result, by trading off a certain amount of accuracy caused by reducing the complexity of the DNN model, significant improvements in both speed and energy efficiency can be achieved. For these reasons, approximate computation is the key to accelerating DNN inference processing on FPGAs.

Table 4.2 and Table 4.3 summarizes the different hardware-oriented optimization methods for approximate computation and illustrates how each of these techniques contributes to the acceleration of DNN inference on FPGAs in terms of reduced computational complexity and memory requirements, respectively.

Table 4.2: Summarization of main hardware-oriented optimization methods for approximate computation to accelerate DNN inference processing on FPGAs.

Method	*Description*	*More details*
Data quantization	Reducing precision of operands and operators to reduce DNN model complexity	Linear/non-linear - Static/dynamic - Binary/ternary - Post-training/QKeras
Network pruning	Reducing redundant DNN model parameters which are not too sensitive to the performance	Fine/coarse pruning Magnitude-/energy-based fine pruning
Non-linearity statistics	Exploiting statistics of activation function to increase the sparsity of DNN model activations	ReLU activation function
Knowledge distillation	Distill knowledge from large DNNs to compact DNN to reduce computational cost	-
Structural simplifications	Designing special structural CONV kernels to save parameters	Pre-/post-training approach

Table 4.3: How each hardware-oriented approximation method helps to accelerate DNN inference processing on FPGAs.

		HW-oriented Optimizations Methods for Approximate Computation		
		Reduce Precision of Operands and Operators	Reduce Memory Size and Bandwidth Requirements	Reduce Number of Operations and Model Size
Data Quantization	Linear Quantization	✓	✓	✗
	Non-linear Quantization	✓	✓	✗
	Binary/Ternary Quantization	✓	✓	✗
Model Compression	Network Pruning	✗	✓	✓
	ReLU Statistics	✗	✗	✓
	Knowledge Distillation	✗	✓	✓
	Structural Simplifications	✗	✗	✓
	Hybrid Strategies	✓	✓	✓

The implementation of these architectural design improvements and hardware-oriented optimization techniques for approximate computation, when combined with fine-tuning of network parameters used in order to restore an appropriate level of accuracy, allow the performance of FPGA-based DNN inference accelerators to be enhanced in terms of both speed and energy efficiency over GP solutions, while maintaining a low design cost and a high level of re-programmability, something that ASIC solutions do not provide. This shows that FPGA is the most promising processing unit for NN acceleration at the edge. In addition, the flexibility of FPGAs allows a large number of other hardware/software optimization methods working at different levels of abstraction to be implemented in conjunction with the approximation methods, such as algorithmic and data-path optimizations, which, although less effective in terms of achievable improvements, allow for additive increases in hardware accelerator performance.

All the techniques for optimizing the development of FPGA-based DNN inference accelerators analyzed are applicable to the implementation of a wide range of ANN types. The analysis carried out focuses mainly on the three types of DNN models, for which FPGAs have emerged as the most promising processing units to accelerate DNN inference processing at the edge, which are deep MLPs and CNNs, which focus on maximizing the performance and accuracy of the model, and SNNs, which are mainly used for low-power ML applications. This is due to the feed-foward

nature of such types of ANNs allows maximizing the exploitation of the inherent parallelism of DNN models at various levels following a pipelined hardware execution available on FPGAs.

In conclusion, by exploiting all the different degrees of freedom offered by FPGAs, a wide range of optimizations can be implemented at the level of both architectural design improvements and hardware-oriented optimization algorithms for approximate computation. Thus, the flexibility and re-programmability of FPGAs allows optimizing hardware implementations of computationally complex DNN models, demonstrating that hardware accelerators based on this type of processing unit represent the best possible compromise for DNN inference processing at the edge, both in terms of high performance and low power consumption.

4.2 Future Directions for FPGA-based DL Accelerators

Currently, most of FPGA-based DNN accelerators only perform inference on-line at the edge, deferring training to a static off-line execution on the cloud, due to its high computational and memory cost.

Although solutions to train complex DNN models directly on embedded devices is attractive for ML applications with stringent speed and energy efficiency constraints, the higher computational power and storage resource requirements of training compared to inference make its execution at the edge a challenging task.

The major problem in implementing training on FPGA-based DNN accelerators lies in the limited availability of resources which clashes with the need to maintain the highest possible level of precision, in order not to reduce the final system accuracy.

Training performed on the cloud not only implies the inability to adjust DNN model parameters dynamically as the task to be performed changes, but also results in increased latency and power consumption required to move the training data and DNN model to be re-trained off-chip to the cloud and, once training is complete, to upload the newly re-trained DNN model to the edge device.

Looking ahead, the future will be to move the training processing performed off-chip at the edge, in order to be able to perform dynamically learning directly on-chip, to adapt the model to local environment, as shown in Figure 4.1.

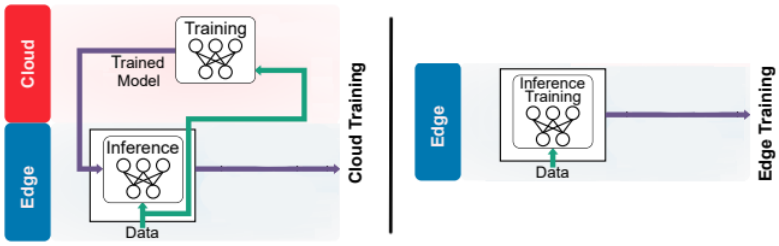

Figure 4.1: Comparison of cloud training (on the left) and edge training (on the right). Source [32]

By leveraging transfer learning, and then using the cloud-trained DNN model as a starting point to perform an edge adjustment, the DNN model parameters can be dynamically fine-tuned on-line. As a result, after edge local training, performing both phases that make up a DL algorithm directly on the embedded device will improve performance in terms speed and energy efficiency due to the elimination of the iteration with the cloud. (see Figure 4.2).

Since DNN processing on edge devices has completely different requirements than on cloud, the main goal of embedded applications is not to focus on achieving high computational performance, but on decreasing hardware resource consumption and improving latency and energy cost. Moving the training execution from the cloud to the edge on FPGAs requires the adoption of some

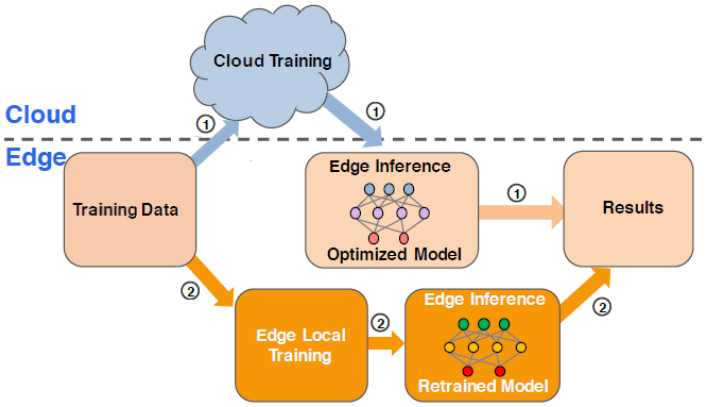

Figure 4.2: Sequence of steps for present ① (i.e. cloud training/edge inference) and future ② (i.e. edge training/ edge inference) DL algorithm processing. Source [67]

optimization techniques that can both improve the management of the limited number of available hardware resources, and reduce the computational and memory cost.

Specifically, data quantization and model compression techniques cannot be applied during training in an aggressive way, due to the fact that they would degrade model performance too much during inference. Thus, much work still needs to be done in terms of both architectural design improvements and the development of optimization techniques to make it possible to run the entire DL algorithm at the edge.

Bibliography

[1] Peter Jeffcock. What's the difference between ai, machine learning, and deep learning.

[2] Safaa Laqtib, Khalid El Yassini, and Moulay Hasnaoui. A deep learning methods for intrusion detection systems based machine learning in manet. In *2015 IEEE Congress*, pages 1–8, 10 2019.

[3] Kamel Abdelouahab. *Reconfigurable hardware acceleration of CNNs on FPGA-based smart cameras*. PhD thesis, Universite Clermont Auvergne, 12 2018.

[4] Ashutosh Nanda. Neural networks concepts.

[5] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[6] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. *Efficient Processing of Deep Neural Networks*. Morgan & Claypool Publishers, 2020.

[7] William G. Wong. Neural-network hardware drives the latest machine-learning craze.

[8] Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2018.

[9] Théodore Bluche. *Deep neural networks for large vocabulary handwritten text recognition*. PhD thesis, Paris 11, 2015.

[10] Ayad Saad Almryad and Hakan Kutucu. Automatic identification for field butterflies by convolutional neural networks. *Engineering Science and Technology, an International Journal*, 23(1):189 – 195, 2020.

[11] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678, 2016.

[12] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of fpga-based neural network accelerator, 2018.

[13] Wikipedia. von neumann architecture.

[14] Rolf Verluis. The mining algo technology progression: Cpus, gpus, fpgas, asics.

[15] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. *IEEE Access*, 10 2006.

[16] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

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

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

[19] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment, 2020.

[20] M. Reljan-Delaney and J. Wall. Solving the linearly inseparable xor problem with spiking neural networks. In *2017 Computing Conference*, pages 701–705, 2017.

[21] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[22] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead. *IEEE Access*, page 1–1, 2020.

[23] Matthew Kay Fei Lee, Yingnan Cui, Thannirmalai Somu, Tao Luo, Jun Zhou, Wai Teng Tang, Weng-Fai Wong, and Rick Siow Mong Goh. A system-level simulator for rram-based neuromorphic computing chips. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):1–24, 2019.

[24] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. Spiking neural networks hardware implementations and challenges: A survey. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(2):1–35, 2019.

[25] Maxence Bouvier, Alexandre Valentian, Thomas Mesquida, Francois Rummens, Marina Reyboz, Elisa Vianello, and Edith Beigne. Spiking neural networks hardware implementations and challenges. *ACM Journal on Emerging Technologies in Computing Systems*, 15(2):1–35, Jun 2019.

[26] Spencer Valancius, Edward Richter, Ruben Purdy, Kris Rockowitz, Michael Inouye, Joshua Mack, Nirmal Kumbhare, Kaitlin Fair, John Mixter, and Ali Akoglu. Fpga based emulation environment for neuromorphic architectures, 2020.

[27] Angel Izael Solis. Dedicated hardware for machine/deep learning: Domain specific architectures. *IEEE Access*, 2019.

[28] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 44(3):367–379, 2016.

[29] Yu-Hsin Chen et al. *Architecture design for highly flexible and energy-efficient deep neural network accelerators*. PhD thesis, Massachusetts Institute of Technology, 2018.

[30] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks. *Nature Electronics*, 1(4):216–222, 2018.

[31] Alex Omø Agerholm. In the age of fpga.

[32] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Muhammad Shafique, Guido Masera, and Maurizio Martina. An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks. *Future Internet*, 12(7):113, 2020.

[33] Vivienne Sze, Yu-Hsin Chen, Joel Emer, Amr Suleiman, and Zhengdong Zhang. Hardware for machine learning: Challenges and opportunities. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8. IEEE, 2017.

[34] Vivienne Sze and Chen. Tutorial on hardware architectures of deep neural networks. In *ISCA*, pages 5–9. IEEE, 2017.

[35] Yudong Tao, Rui Ma, Mei-Ling Shyu, and Shu-Ching Chen. Challenges in energy-efficient deep neural network training with fpga. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 400–401, 2020.

[36] BiXBit. Fpga for mining: what trends will prevail in 2019.

[37] Francesco Conti, Pasquale Davide Schiavone, and Luca Benini. XNOR neural engine: a hardware accelerator IP for 21.6 fj/op binary neural network inference. *CoRR*, abs/1807.03010, 2018.

[38] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. Accelerating cnn inference on fpgas: A survey, 2018.

[39] Griffin Lacey, Graham W Taylor, and Shawki Areibi. Deep learning on fpgas: Past, present, and future. *arXiv preprint arXiv:1602.04283*, 2016.

[40] Aysegul Dundar, Jonghoon Jin, Vinayak Gokhale, Berin Martini, and Eugenio Culurciello. Memory access optimized routing scheme for deep networks on a mobile coprocessor. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[41] Blair Cook. Vlsi design: Fpga technology.

[42] 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*, pages 161–170, 2015.

[43] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[44] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of FPGA based neural network accelerator. *CoRR*, abs/1712.08934, 2017.

[45] Erwei Wang, James J. Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter Y. K. Cheung, and George A. Constantinides. Deep neural network approximation for custom hardware: Where we've been, where we're going. *CoRR*, abs/1901.06955, 2019.

[46] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823–7859, 2018.

[47] Donny Chow. Fixed point vs. floating point illustration.

[48] Marco Baldi, Nicola Maturo, Enrico Paolini, and Franco Chiaraluce. On the use of ordered statistics decoders for low-density parity-check codes in space telecommand links. *EURASIP Journal on Wireless Communications and Networking*, 2016, 11 2016.

[49] Claudionor N. Coelho Jr. au2, Aki Kuusela, Shan Li, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers. Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders, 2020.

[50] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal brain damage. In *Proceedings of the 2nd International Conference on Neural Information Processing Systems*, NIPS'89, page 598–605, Cambridge, MA, USA, 1989. MIT Press.

[51] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the 4nd International Conference on Neural Information Processing Systems*, pages 806–814, 06 2015.

[52] Pavlo Molchanov, Timo Aila Stephen Tyree, Tero Karras, and Jan Kautz. Pruning convolutional neural networks.

[53] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[54] Vivienne Sze. Approaches for energy efficient implementation of deep neural networks.

[55] Vivienne Sze. How to evaluate efficient deep neural network approaches.

[56] Moazze Mhossain. Easy way to understand convolutional neural network.

[57] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[58] Adriana Romero, Samira Ebrahimi Kahou, Polytechnique Montréal, Y. Bengio, Université De Montréal, Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. In *in International Conference on Learning Representations (ICLR*, 2015.

[59] Jianping Gou, Baosheng Yu, Stephen John Maybank, and Dacheng Tao. Knowledge distillation: A survey. *arXiv preprint arXiv:2006.05525*, 2020.

[60] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.

[61] 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, 2017.

[62] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network, 2014.

[63] Aili Wang, Minhui Wang, Kaiyuan Jiang, Mengqing Cao, and Yuji Iwahori. A dual neural architecture combined squeezenet with octconv for lidar data classification. *Sensors*, 19(22):4927, 2019.

[64] Sik-Ho Tsang. Mobilenetv1: Depthwise separable convolution (light weight model).

[65] Simon Low. Squeezenet and mobilenet: Deep learning models for mobile phones, 2018.

[66] Matt Eding. Data structures - sparse matrices.

[67] Xingzhou Zhang, Yifan Wang, Sidi Lu, Liangkai Liu, Lanyu Xu, and Weisong Shi. Openei: An open framework for edge intelligence, 06 2019.