Università degli Studi di Cagliari

Ph.D. DEGREE

ELECTRONIC AND COMPUTER ENGINEERING

Cycle XXXIV

TITLE OF THE Ph.D. THESIS

ACCELERATION OF ARTIFICIAL NEURAL NETWORKS AT THE EDGE: ADAPTING FLEXIBLY TO EMERGING DEVICES AND MODELS

Scientific Disciplinary Sector(s)

ING-INF/01

Ph.D. Student: MARCO CARRERAS

Supervisor LUIGI RAFFO

Co-Supervisor PAOLO MELONI

Final exam. Academic Year 2020/2021

Thesis defence: April 2022 Session

Università degli Studi di Cagliari
Dept. of Electrical and Electronic Engineering
PhD program in Electronics and Computer Engineering

Acceleration of Artificial Neural Networks at the edge: adapting flexibly to emerging devices and models

Marco Carreras

Supervisor: Prof. Luigi Raffo
Co-Supervisor: Dr. Paolo Meloni
Ph.D. Coordinator: Prof. Alessandro Giua
Scientific Disciplinary Sector: ING-INF/01

XXXIV Cycle
Final Exam Academic Year 2020/2021
Thesis Defence: 20th April 2022

Abstract

Convolutional Neural Networks (CNNs) are nowadays ubiquitously used in a wide range of applications. While usually CNNs are designed to operate on images for computer vision (CV) tasks, more recently, they have been applied in multiple other embedded domains, to analyze different information and data types.

A key research topic involving CNNs is related to methodologies and instruments implementing a shift from cloud computing to the edge computing paradigm. The classic implementation of CNN-based systems relies on the cloud: an embedded system samples data acquired by adequate sensors and sends them to a remote cloud computing facility, where the data is analyzed on high-performance processing platforms. However, to really enable ubiquitous use of CNNs, some use-cases require moving the classification/recognition tasks at the edge of the network, executing the CNN inference near-sensor, directly on embedded processing systems. At-the-edge data processing has multiple potential benefits: it improves responsiveness and reliability, avoids disclosure of private information, and reduces the communication bandwidth requirements posed by the transmission of raw sensor data.

Among the possible technology substrates that may be used to implement such embedded platforms, a widely used solution relies on processing systems integrating Field Programmable Gate Arrays (FPGAs). The Digital Signal Processing (DSP) slices available in modern FPGAs are very well suitable for the execution of multiply-and-accumulate operations, representing the heaviest workload in CNNs. In particular, All-Programmable Systems on Chip (AP-SoCs), i.e. heterogeneous processing systems designed to exploit the cooperation between general-purpose processing cores and FPGA resources, can accommodate quite effectively both the highly parallel data-crunching operations in the network and the other more control-like and housekeeping-related actions surrounding them within the overall software applications.

The work in this thesis focuses on CNN inference acceleration on AP-SoCs. It starts from a reference architecture, an FPGA-based CNN inference accelerator named NEURAghe [73], and extends it to assess its flexibility to different target devices and its applicability to a wider range of design cases and network topology.

To this aim, in the first phase of the work, we have aggressively parameterized the architecture, to be capable of shaping it into different configurations to be implemented on various device sizes.

In a second phase, we have tested and studied modifications to extend NEURAghe's approach from mainstream CNNs, whose execution is widely supported by multiple accelerators in literature, to less deeply explored algorithm flavours, namely:

- Temporal Convolutional Network (TCN), operating with mono-dimensional dilated kernels on sequences of samples;
- Depthwise separable convolutions, that reduce the number of Multiply-Accumulate operations (MACs) to be performed per layer and, consequently, if countermeasures are not taken, reduce the utilization rate of hardware MAC modules in NEURAghe;
- Event-based Spiking Neural Networks (SNNs), that requires an entirely different architecture pattern, that needs to be finely tuned and integrated into the NEURAghe system template to be effectively used on FPGA;

From the first phase results, it is possible to show how a parametrized architecture could lead to different configurations suitable to different scenarios and adaptable to different target devices by trading-off between performance in terms of GOPS/s, target device cost, and power consumption.

The activities towards TCN support show how the same architectural pattern used in NEURAghe could be adapted to cope with the peculiarities of these kinds of networks, additionally increasing the overall flexibility. The resulting architecture extends its processing capabilities with the introduction of the support to freely selectable parameters and a dedicated processing scheduling which enable TCN execution. Results show that the architecture is capable to reach up to 96% of efficiency, in a specific TCN use case, considering the highest GOPS/s rate with respect to the peak achievable by the configuration. These improvements enhance also classical CNNs execution when dealing with irregular network patterns showing an improvement of 40% in terms of execution time in the considered experiments.

The exploration concerning the introduction of the Depthwise separable support in NEURAghe with a non-invasive approach, suggests that it is worth reconsidering the standard convolutional scheme while dealing with these kinds of operators to face its otherwise inevitable inefficiency given by the partial exploitation of the reconfigurable logic processing capabilities. Results show that re-designing the accelerator to offer specific support to these operators also by changing the data processing pattern, improves the efficiency with respect to non-invasive architectural solutions with regard to NEURAghe.

Finally, the integration of an SNN accelerator in FPGA pursued by applying the NEURAghe model also to Spiking Neural Networks sets the basis for the development of a topology agnostic architecture results in an execution timing comparable with a referenced work in literature allowing to be more flexible towards different layer characteristics without reconfiguration.

Contents

List of Figures **vi**

List of Tables **ix**

Introduction **1**

1 Context **1**

1.1 Edge-computing paradigm 1
1.2 Convolutional Neural Networks 4
1.3 Accelerating CNN inference on FPGA: a SoA overview 7
1.4 Reference Architecture: NEURAghe 13
1.4.1 Convolution Specific Processor 14
1.4.2 Execution scheduling 17

2 Exploring device parameterisation **19**

2.1 Parameter exploration 19
2.2 Experimental results 22
2.3 Comparison and summary 25

3 TCN inference optimization on FPGA-based accelerator **26**

3.1 TCN model generalities 28
3.1.1 Mono-dimensional dilated convolution 28
3.2 TCN supporting hardware features 30
3.2.1 Freely selectable kernel sizes 31
3.2.2 Flexible activations and weights fetching 32
3.2.3 TCN support in firmware 35
3.2.4 Improving through batch processing 36
3.3 Hardware Implementation Evaluation 38
3.3.1 Design Space Exploration 38
3.3.2 Implementation on different SoCs 38

3.4 Experimental Results 41
3.4.1 ECG classification use-case 43
3.4.2 Res-TCN use-case 46
3.4.3 WN-PNT use-case 48
3.4.4 Assessment of hardware vs. software speed-up 50
3.4.5 Comparison to other APSoC based accelerators 52
3.5 Summary 53

4 Adapting for mobile network operators 54
4.1 Depthwise Separable Convolution and MBConv 55
4.2 Exploring unobtrusive solutions 57
4.3 Specific support for lightweight operators 63
4.4 Summary 66

5 Spiking Neural Engine to FPGA adaptation 67
5.1 Spiking Neural Engine 69
5.2 Mapping to FPGAs 71
5.2.1 Exploiting NEURAghe infrastructure 74
5.2.2 Implementation Results 75
5.3 Experimental Results 76

6 Conclusions 79

List of Figures

1.1 Convolutional Layer: a $KS \times KS \times I_CH \times O_CH$ filter tensor applied to an $IW \times IH \times I_CH$ input tensor produces an $OW \times OH \times O_CH$ output tensor . . . 5
1.2 Convolutional operator applied to a 2D feature with a KS=3 kernel size and zero padding and stride unitary. . . . 6
1.3 Pooling operator application and ReLU with function $f(x) = max(0, x)$ . . . 6
1.4 NEURAghe architecture . . . 13
1.5 Convolution Engine. $N_{rows} \times N_{cols}$ MAC Matrix . . . 15
1.6 Linebuffer structure . . . 16
1.7 Single and Multi-trellis SoP cascade . . . 17
1.8 Scheduling scheme . . . 18

2.1 NEURAghe architectural template with multiple CPSs . . . 20
2.2 Roofline model for the 4×4 (blue) and 2×2 (red) configurations. The vertical dashed line represents the average operational intensity of a convolutional layer. . . . 23
2.3 Comparison of the performance efficiency, achieved on all the convolution layers (L) in ResNet-18 and VGG-16, by different configurations, estimated as the ratio between actual performance (GOps/s) and peak performance (GOps/s). Efficiency for the double cluster configuration on ResNet-18 is not reported as it is mainly limited by the GPP . . . 23

3.1 TCN execution graph. For this example, see (3.2), $receptivefield = 1 + (2 - 1) \times 1 + (3 - 1) \times 2 + (4 - 1) \times 3 = 15$ . . . 29
3.2 NEURAghe architecture with double weight DMA . . . 30
3.3 SoP elaboration scheme . . . 31
3.4 MAC Matrix (a) and Sum of Product Unit (b) . . . 32

3.5 CE memory transfers configuration. Communication between memory regions and CE is made by means of configurable source modules which have access to BRAM modules through the exposed B-ports. Each source module can be programmed according to layer characteristics and it is able to feed each SoP Unit with 4 input samples/cycle from the Activation Memory Region and weight kernel elements from Weight Memory Region, both composed by independently accessible memory banks. Moreover, with the same behaviour, a source and a sink module handle transfers from Output Memory Region by feeding Shift Adder Modules with partially computed values from the previous step and storing actual results. . 33
3.6 BRAM read conflict example. Samples are stored in BRAM modules using interleaving. Consecutive samples are stored in adjacent RAMB18 banks. 4 DSP slices in a SoP unit, with stride 2, in the first cycle of the convolution, load respectively samples 0,2,4,6. With four RAMB18 banks, samples 0-4 and 2-6 are on the same bank, creating a conflict. The bottom part shows how the conflict is avoided by doubling the number of banks. 34
3.7 TCN execution on NEURAghe: *batch size* = 1 35
3.8 Use case benchmark's Roofline Model for the Convolution Specific Processor (12x4 MAC Matrix in Z-7020 SoC) with respect to different batch sizes 36
3.9 TCN execution on NEURAghe: *batch size* = B 37
3.10 Efficiency trend on ECG [35] for different batch sizes for NEURAghe 12×4 and 11×5 MAC matrix configurations in XC7Z020 and 9×10 MAC Matrix configuration in XCZU3EG 44
3.11 Execution Time comparison on ECG [35] for NEURAghe (a): 12×4 and 11×5 MAC matrix configurations in XC7Z020 and (b): 9×10 MAC Matrix configuration in XCZU3EG 45
3.12 Efficiency trend on Res-TCN [62] for different batch sizes for NEURAghe 12×4 and 11×5 MAC matrix configurations in XC7Z020 and 9×10 MAC Matrix configuration in XCZU3EG 46
3.13 Execution Time comparison on Res-TCN [62] for different batch sizes for NEURAghe 12×4 and 11×5 MAC matrix configurations in XC7Z020 and 9×10 MAC Matrix configuration in XCZU3EG . . . 47
3.14 Efficiency trend on WN-PNT [71] for different batch sizes for NEURAghe 9x10 matrix configuration in XCZU3EG 48
3.15 Execution Time on WN-PNT [71] for different batch sizes for NEURAghe 9x10 matrix configuration in XCZU3EG 49

3.16 Execution time and power efficiency comparison between software execution on a Cortex-A53 quad-core and NEURAghe (XCZU3EG). 51

4.1 Depthwise Separable Convolution 55
4.2 NEURAghe DW phase handling 57
4.3 Exploiting depthwise separable reuse in NEURAghe: full matrix . . 58
4.4 Exploiting depthwise separable reuse in NEURAghe: DW sub-matrix 60
4.5 Efficiency trend and execution time on Mobilenet-V2 MBConv layers 62
4.6 Architecture Overview 63
4.7 Convolution Engine Organization 64
4.8 Efficiency trend and execution time on Mobilenet-V2 MBConv layers processed with the specific engine 65

5.1 SNE architecture overview 69
5.2 SNE Neuron data-path 71
5.3 SNE Neuron Datapath: Ops to DSP mapping 73
5.4 SNE integration within the NEURAghe infrastructure 74
5.5 SNE latency trend with respect to the activity rate 77

List of Tables

2.1 Resource occupation on different configurations 22
2.2 Performance evaluation for different MAC Matrix configurations . . 24
2.3 Performance evaluation for multiple CSPs 24
2.4 Comparison with alternatives in literature 25

3.1 RAMB18 and DSP utilization in NEURAghe architecture with respect to the MAC Matrix shape 39
3.2 Resource occupation on a Xilinx XC7Z020 (12x4 MAC matrix) . . . 40
3.3 Resource occupation on a Xilinx XC7Z020 (11x5 MAC matrix) . . . 40
3.4 Resource occupation on a Xilinx XCZU3EG (9x10 MAC matrix) . . 40
3.5 Implementation related on-chip memory capabilities and use-case networks memory footprint 42
3.6 Convolutional Layer characteristics for Network ECG [35] 43
3.7 Convolutional Layer characteristics for Res-TCN [62] 46
3.8 Comparison between NEURAghe version describend in this chapter (NEURAghe TCN), the previous version (Chapter 2) and other works on ResNet-18 and VGG-16. Xilinx XC7Z020 52

4.1 MobileNet-V2 MBConv layer characteristics 61

5.1 SNE Neurons occupation before (a) and after (b) MAC ops mapped on DSPs 72
5.2 SNE 8x16 (a) and 2x16 (b) architecture synthesis (Xilinx XCZU3EG) . . 73
5.3 SNN accelerator architecture resource occupation (XCZU3EG): without (60 MHz) and with (85 MHz) critical path optimization . . 75
5.4 [30] convolutional SNN architecture 76
5.5 A quantitative comparison between this work and [30]. 77

Introduction

Deep Convolutional Neural Networks have become the go-to solution for a wide range of AI-related problems. Thanks to their outstanding results, they represent the state-of-the-art in image recognition [43], [65], [113], face detection [96], speech recognition [42] and text understanding [109], [110] among other tasks. The widespread use of Convolutional Neural Networks (CNNs), combined with the high computational load typically associated with their execution, has led researchers to extensively work on the development of hardware accelerators for CNN inference [121][17][9][19]. The availability of this kind of hardware is key in embedded use-cases involving near-sensor processing of data, according to the edge computing paradigm [104]. Among the different solutions available in the literature, an important role is played by Field Programmable Gate Array (FPGA) based architectures, and, in more detail, by solutions exploiting the cooperation between general-purpose processors and FPGAs available in modern All-Programmable Systems on Chip (APSoCs), e.g. Zynq-7000 and Zynq Ultrascale+ devices by Xilinx, that take profit from the efficient implementation of Multiply-And-Accumulate (MAC) operations on a large amount of Digital Signal Processing (DSP) Slices available [75].

Moreover, over the years, the diffusion of CNN has meant differentiation, entailing a wide range of approaches, kernel shapes and techniques, opening the way for the second-generation of CNN algorithms. Several methods have been developed, alongside classical approaches, that, for example, differ in the way they apply filters or in the way they deal with input features during the convolutional step, giving also their name to the particular computational step. This is the case of the dilated convolution [91] and the deconvolution (or transposed convolution) [119].

Modern techniques subvert the general trend of making deeper and more complicated networks pursuing higher accuracy, as these advances are not necessary when targeting edge-oriented applications where tasks need to be carried out in a timely fashion on a computationally limited platform and architectures need to be more efficient with respect to size and speed ([48]) as it happens in the embedded domain. This class of networks are known as mobile networks and they often rely

on the Depthwise Separable Convolution operator [92].

Another CNN use case can be found when it comes to dealing with time sequence modelling, a task historically associated with Recurrent Neural Networks (RNNs) [34]. To cope with their training complexity ([14]), over time, two major alternatives have become widespread, namely Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) which try to reduce RNN complexity. However, in contrast with the predominant sequence modelling/RNN association, a very promising alternative approach is based on Temporal Convolutional Networks (TCNs) [13], namely, mono-dimensional CNNs characterised by different kernel sizes and dilation parameters. These networks have proven to outperform generic recurrent architectures such as LSTMs and GRUs opening the way for an investigation on the acceleration of their execution in the embedded domain, as it happened with classical CNNs.

Pursuing the energy efficiency requirement while dealing with deep learning inference at the edge, many accelerators take advantage of the diffusion of quantized networks.

On the other hand, a more recent approach that changed the landscape of the algorithms is the introduction of a promising third-generation of Artificial Neural Networks (ANNs), namely the Spiking Neural Networks (SNNs). They have the benefit of an event-based computation, i.e. the execution takes place only when a new event is detected thus reducing the total number of operations required and allowing an energy proportional data processing. In these networks, neurons feature an internal state, updated over time. Thus inputs and outputs of each layer are binary spikes that indicate when this state has crossed a threshold. Among different neuron models, varying in complexity, the Leaky-Integrate and Fire [54] simplifies hardware design allowing to build SNNs with convolutional layers comparable with those found in CNNs with the additional time dimension. The characteristics of SNNs adapt well to new sensing devices which have evolved in the direction of an efficient data transmission that minimizes power consumption while remaining responsive to external changes as in the case of event-based sensors like Event vision sensors (EVSs).

The proposed Ph.D. thesis moves in such a multi-faceted context, aiming to explore how a flexible adaptation of the edge-oriented hardware acceleration task could be carried out to target the above-mentioned classes of Artificial Neural Networks.

In particular, the focus lies on the category of FPGA-based inference accelerators, starting from the architectural template deployed in the Electric and Electronic Engineering Department of the University of Cagliari in collaboration with the Digital Circuits and Systems Laboratory of ETH Zurich, called NEURAghe. Its original approach lies in the cooperation between the ARM-based processing

system and programmable soft-core integrated into the Convolution Specific Processor implemented in the reconfigurable logic. In this way, the ARM cores are released from the majority of the supervision duties enabling a finer accelerator control.
The main contribution described in this thesis are:

- The exploration of the capabilities offered by taking advantage of the reference architecture flexibility when it comes to adapting to different computational scenarios regarding the traditional CNN inference task. The architecture, aggressively parameterized, has been tested with different use-cases and devices that differ for cost, sizes and resources availability. The results are shown in Chapter 2.

- An in-depth analysis of how such an architectural template could be improved to be adaptable towards any kernel size and dilation rate parameters of bi-dimensional convolution network architectures, but in particular how it can be enabled to execute mono-dimensional convolutional networks (TCN) considering different timing constraints. A methodology for the optimal execution/scheduling of data transfers that takes advantage of the specific sequence-based structure of data in TCNs and a methodology for improving efficiency based on sequence buffering are also presented. This analysis is shown in Chapter 3.

- An evaluation of the efficiency drawbacks for a non-invasive Depthwise separable support introduction while maintaining the same architectural scheme and conversely the necessity to adopt a specific hardware configuration to obtain an advantageous execution. A comparison between results offered by different strategies shows the subsequent performance improvement in terms of efficiency with respect to the theoretical peak performance achievable, justifying the choice made. The discussion can be found in Chapter 4.

- The FPGA implementation of an Application Specific Integrated Circuit (ASIC) conceived Spiking Neural Network Engine by exploiting the NEURAghe architectural infrastructure to speed up the integration process. After a required mapping of some processing and memory elements onto the FPGA slices, we will analyze the resource utilization constraints that emerge to let the architecture fit into a specific target board. Performance results in terms of execution latency of the obtained architecture will be analysed with respect to different SNN activity rates (Chapter 5).

1 | Context

This chapter will provide the basics to contextualise the scope of this thesis work.

Firstly, we identify how the work presented in this thesis collocates in the multifaceted field of edge computing. Then we describe the Convolutional Neural Networks salient characteristics, with a particular focus on the convolutional operator. Then an overview of the CNN inference accelerator landscape will be given. Lastly, a thorough description of the reference accelerator for this work will be provided.

1.1 Edge-computing paradigm

Over the past few years, the Edge computing paradigm has gained popularity as it provides low latency, mobility, location awareness, proximity to the user and security support to delay-sensitive applications and in applications where a near-sensor/near-data elaboration is required. The name is given for the way computing power is brought to the "edge" of a device or network, more precisely it is moved closer to where data is generated, usually a sensor. By processing data at a network's edge, there is a reduction of the large amounts of data travelling between servers, the cloud, and devices or edge locations. This solves the infrastructure issues found in conventional data processing, such as latency and bandwidth. A recent survey by Khan et al. gives an overview of the edge computing paradigm landscape by identifying three main sub-categories, namely Cloudlets, Fog computing and Mobile Edge computing [60].

The edge computing paradigm enables also faster data elaboration and sovereignty which is particularly important for modern applications such as data science and AI.

However, edge computing should not only be considered as a computation offload of a certain workload from a centralized data centre to different nodes in the network. It also covers a multifaceted landscape of different cases, including self-exhaustive applications with limited data transmission over the network, like, for example, AI vision applications where data security is mandatory (self-driving

cars, healthcare).

This allows the edge computing paradigm definition to enclose a wide range of applications as general Internet of Things (IoT) and AI-related tasks with different degrees of computational load and also a broad landscape of target devices which differ mostly in technology (i.e. Application Specific Integrated Circuit, ASIC or Field Programmable Gate Array, FPGA), power consumption, cost and form factor.

IoT applications usually demand for relatively simple, ultra-low power and low-cost devices with a tiny form factor. They usually leverage custom or general-purpose processing units with limited capabilities which expose a task-related interface to the environment (sensors). The budget of energy consumption ranges from μW to mW [72, 26]. Nowadays, IoT devices face also AI-related problems implementing also an in-place data processing that must be done in a highly optimized way to fit in the device [107]. It is the case of some lightweight nodes for healthcare monitoring [86].

When it comes to supporting AI applications by providing a significant degree of computational power at the edge it is possible to employ different devices depending on the specific use case. Despite this, among their common characteristics, there are a small form factor and limited power consumption. As an example, NVIDIA gives an overview of their AI edge-oriented devices from the Jetson family [4]. They mostly exhibit a power consumption from $5W$ to $20W$, reaching higher values, up to $50W$, in a few cases.

Given these baseline characteristics, as said, different hardware solutions could be taken into account. A chip-down development, where a custom circuit board is designed from scratch, has its advantages as it will result in a highly optimized device. On the other hand, it can take significant development time and cost to reach production readiness. Moreover, the pace of AI innovation is incredibly rapid. Fixed silicon devices that implement AI can quickly become obsolete due to the emergence of newer, more-efficient AI models.

One of the most promising technologies, for AI-enabled edge applications, is adaptive computing [2]. It includes reconfigurable hardware such as FPGAs, highly optimisable for specific applications, which in modern Systems-on-Chip (SoCs) are coupled with multi-core CPUs. This configuration allows rapid and flexible deployment of a hardware-tailoring application while ensuring adaptation when needed. Additionally, FPGAs usually provides a good compromise between power consumption and performance given by the high level of parallel and pipelined computation available.

The work presented in this thesis moves in the field of edge-oriented AI applications, developed in modern MPSoCs leveraging the cooperation between an FPGA and a general-purpose processor. More precisely we focus on a sub-class

of the Deep Learning algorithms, namely the class of Artificial Neural Network known as Convolutional Neural Networks (CNNs) implemented on Xilinx devices belonging to the Zynq-7000 and Zynq Ultrascale+ families.

1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a class of deep feed-forward brain-inspired Artificial Neural Networks (ANN) where data propagate and are processed in a unique direction from the input to the outcome. These networks can generically be represented as directed graphs where edges are represented by data tensors, and each node represents an operation (a layer) transforming one or more inbound tensors into an outbound tensor. Most often the data tensors considered in CNNs for image processing applications are three-dimensional, with one dimension representing different channels or feature maps plus two spatial dimensions. In the final layers of a CNN, some of these tensors can be "collapsed" to 1D vectors, where the spatial notion is lost. Operations commonly performed in a node are convolutions, followed by a non-linear activation (often rectification), pooling, and fully connected layers.

Convolution

Convolutional layer transforms a 3D tensor of size $IW \times IH \times I_CH$ into a new 3D tensor of size $OW \times OH \times O_CH$, by applying the convolution operator between the tensor given as input to the layer and the filter tensor of size $KS \times KS \times I_CH \times O_CH$. IW, IH, OW and OH are the input and output spatial feature sizes respectively, while I_CH and O_CH are input and output channels and KS is the kernel size. Figure 1.1 highlights these concepts.

Usually, the tensor obtained after passing through a convolution layer is called feature map or *activation* map as the convolution aims to extract some kind of features from the original input thus becoming an abstract representation of it. Feature extraction is made possible by the filter tensor, a vector of *weights* kernels applying a specific function to the values that belong to the receptive field of the input feature. The receptive field is usually an area of square values of size $KS \times KS$. Weight values are learned during the training process.

Equation 1.1 gives a mathematical representation of the convolution operator: $\mathbf{W}$ is the tensor of weights, $\mathbf{b}$ is the one of biases (generally one bias per O_CH is summed), $\mathbf{x}$ and $\mathbf{y}$ are input and output tensors. k_i and k_o indexes select among tensor features and the corresponding kernel among weight tensor.

$$k_o \ \in 0...O_CH-1, \ \mathbf{y}(k_o) = \mathbf{b}(k_o) + \sum_{k_i=0}^{I_CH-1} \mathbf{W}(k_o, k_i) \otimes \mathbf{x}(k_i) \tag{1.1}$$

Figure 1.2 gives a 2D demonstration of how the convolution works in CNN. The stride parameter represents the amount of shifting done by the kernel while

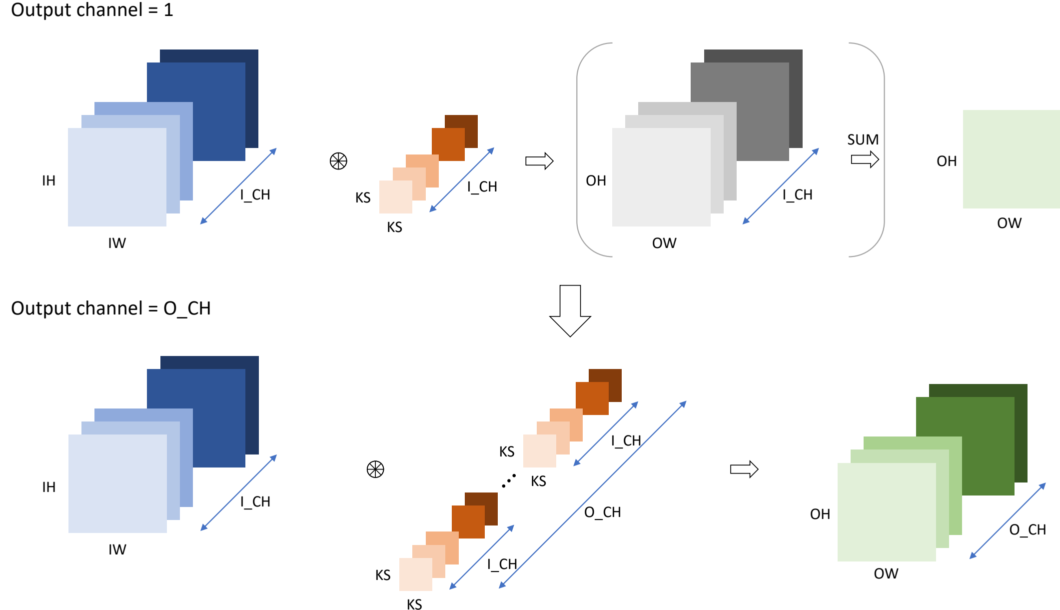


Figure 1.1: Convolutional Layer: a $KS \times KS \times I_CH \times O_CH$ filter tensor applied to an $IW \times IH \times I_CH$ input tensor produces an $OW \times OH \times O_CH$ output tensor

sliding the input feature; the zero-padding parameter can be seen as a zero frame applied around the feature to preserve the same size in output, it is usually $\frac{KS-1}{2}$.

ReLU and Pooling

As said, convolution is often followed by a non-linear activation. The most commonly used are rectifiers or ReLU (Rectifier Linear Unit) which, for example, apply a ramp function to each output feature sample (Figure 1.3b).

Pooling layers operate a feature map down-sampling by sliding a 2D filter in order to summarize the presence of features in patches lying within the region covered by the filter. Commonly used are max pooling (Figure 1.3a) and average pooling.

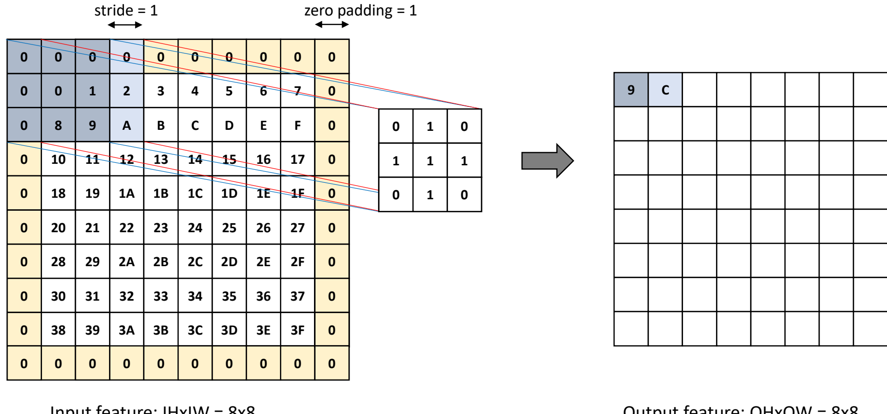


Figure 1.2: Convolutional operator applied to a 2D feature with a KS=3 kernel size and zero padding and stride unitary.

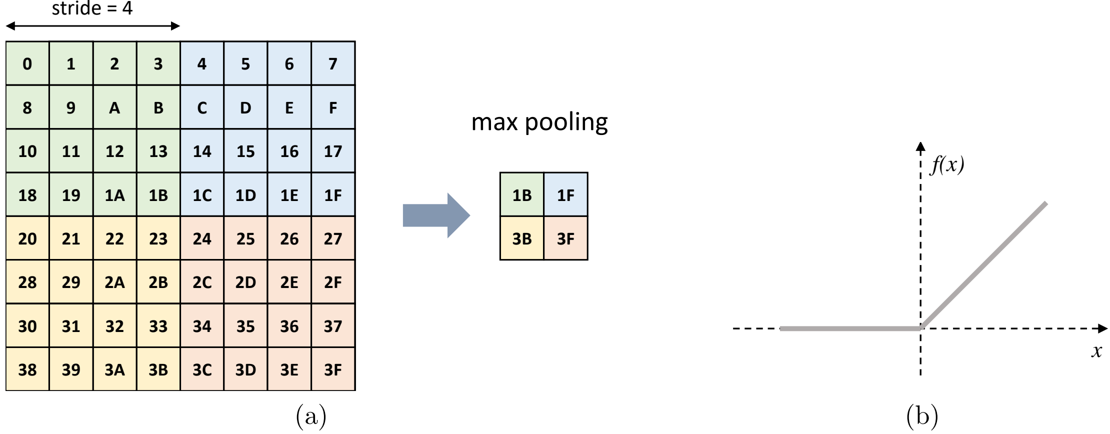


Figure 1.3: Pooling operator application and ReLU with function $f(x) = max(0, x)$

Fully Connected Layer

Fully connected layers are the final layers to be applied. In this case, every neuron produced depends on every input neuron from the input tensor which usually is flattened. The mathematical structure is similar to the convolution one, but they operate on 1D vectors resulting in a matrix-vector multiplication:

$$\mathbf{y} = \mathbf{b} + \mathbf{W} \cdot \mathbf{x} \tag{1.2}$$

1.3 Accelerating CNN inference on FPGA: a SoA overview

The landscape of FPGA-based accelerators for CNN is crowded and multifaceted [39]. Several approaches have been proposed in recent years, focusing both on the embedded domain and on architectures aimed to speed-up execution on cloud servers. However, work on FPGA acceleration has mainly focused on classic CNN networks.

Convolutional Neural Network accelerators

Yu et al. [117] developed an FPGA acceleration platform that leverages a unified framework architecture for general-purpose CNN inference acceleration at a data centre achieving a throughput comparable with the state-of-the-art GPU in this field, with smaller latency.

Zhang et. al. [120] proposed Caffeine, a hardware/software library to efficiently accelerate CNNs on FPGAs, leveraging a uniformed convolutional matrix multiplication representation.

Ma et. al. [70] presented a Register-Transfer level (RTL) CNN compiler that automatically generates customized FPGA hardware for the inference tasks of CNNs to enable fast prototyping from software to FPGA.

These frameworks provide huge performance gains when compared to state-of-the-art accelerators and general-purpose CPUs and GPUs. However, they leverage large FPGA devices such as Virtex7 and Arria 10, mainly targeting server applications exploiting batching to improve memory access performance and bandwidth utilization.

When it comes to design cases in the embedded domain, slightly different objectives need to be taken into account. First, cost and power budget, thus the target setup usually involves smaller FPGAs with less processing and storage resources. Second, latency is an important metric. Images/samples under classification/recognition belong to a single acquisition stream and batch size must be very limited, often constrained to one. Third, systems are usually required to be autonomous, thus designers often focus on etherogeneous systems-on-chip, empowered by processing systems in charge of data management, network connectivity and other housekeeping tasks independently. Thus, closer to this work, there are more resource-constrained approaches, mostly validated on smaller devices, e.g. Zynq APSoCs, usually XC7Z045 or smaller, or Zynq Ultrascale+ MPSoCs.

Venieris et. al. [100] presented a latency-driven design methodology for mapping CNNs on FPGAs. As opposed to previously presented approaches mainly intended for bandwidth-driven applications, this work targets real-time applica-

tions, relying on Xilinx high-level synthesis tools for mapping (i.e. Vivado HLS), demonstrated on relatively simple CNN such as AlexNet, and a very regular one such as VGG16 featuring only 3×3 kernels, providing a peak performance of 123 GOps on a Xilinx XC7Z045 SoC.

Some recent more structured works provide a complete framework to help design CNNs into FPGA. hls4ml [29] has the objective of translating machine learning algorithms for FPGA implementation using Python. It is based on an open-source code design workflow supporting network pruning and quantization also towards low-power inference implementations. Their support is extended over mono and bi-dimensional CNNs [3]. FINN-R [15] represents the evolution of FINN presented in [98]. It automates the creation of fully customized inference engines on FPGAs especially targeting Quantized Neural Networks (QNNs) from training to deployment. Both hls4ml and FINN-R leverages High-Level Synthesis (HLS) tools.

Despite the unquestionable value of these works, they leverage an automated flow based on HLS and network graph transformation which transfer the entire network into the device, thus requiring re-programmation for different use-cases.

Other works focus on a template-based approach based on programmable or customizable RTL accelerators [70][33][81]. This is also the context in which it moves this thesis, referring to ready-to-use architectural templates controlled directly by software-APIs and designed to deal with more standard data formats (8 and 16 bits) so that pre-trained networks can be executed with good accuracy without re-training.

SnowFlake [33] exploits a hierarchical design composed of multiple compute clusters. Each cluster is composed of four vectorial compute units including a vectorial Multiply-Accumulate (MAC), vectorial max, a maps buffer, weights buffers and trace decoders. SnowFlake provides a computational efficiency of 91%, and an operating frequency of 250 MHz (best-in-class for CNN accelerators on Xilinx XC7Z045 SoC). However, although the vector processor-like nature of the accelerator is very flexible, delivering significant performance also for 1×1 kernels, it prevents fully exploiting the spatial computation typical of application-specific accelerators, which leads to overheads due to load/store operations necessary to fetch weights and maps from the buffers. This is highlighted by the low utilization of the DSP slices available on the FPGA (i.e. only 256 over 900), and by the performance when executing end-to-end convolutional neural networks, which is lower than that of other architectures including the template referenced by this work even though the operating frequency of the CNN engine is significantly higher.

Several approaches tackling FPGA architectures for image-processing CNN have explored the reduction of the precision of arithmetic operands to improve energy efficiency. Although most of the architectures available in literature feature a

precision of 16-bit (fixed-point)[100, 33, 70] numerous reduced-precision implementations have been proposed recently, relying on 8-bit, 4-bit accuracy for both maps and weights, exploiting the resiliency of CNNs to quantization and approximation [81].

Qiu et. al. [81] proposed a CNN accelerator implemented on a Xilinx XC7Z045 platform exploiting specific hardware to support 8/4 bit dynamic precision quantization, at the cost of 0.4% loss of classification accuracy. Other extreme approaches to quantization exploit ternary [80] or binary [98] neural-networks accelerators for FPGA. This approach significantly improves the computational efficiency of FPGA accelerators, allowing to achieve performance levels as high as 8 TOPS [80].

Recent work by Rasoulinezhad et al. [82], starting from the Xilinx DSP slices, proposed an optimized DSP block called PIR-DSP to efficiently map 9, 4 and 2 bits data precision MAC operations. It is implemented as a parameterized module generator targeting both FPGAs and Application Specific Integrated Circuits (ASICs) reaching an estimated run time energy decrease of up to 31% for a MobileNet-v2 implementation compared with a standard DSP mode. Other works, like Wang et al. [105], leverage FPGA Look-Up Table (LUT) blocks as inference operators for Binary Neural Network (BNN) achieving up to twice the area efficiency compared to state-of-the-art binarized NN implementations and against several standard network models.

Moreover, while extremely reduced precision networks can easily reach good classification accuracy on small datasets, like MNIST, CIFAR10, SVHN, GTSRB, training quantized versions of larger networks, such as VGG or ResNet [23], capable of dealing with more complex datasets and tasks, is still a big challenge. The usability of extreme quantization is also not demonstrated for TCN-related tasks, sequence classification and modelling.

This thesis focuses on NEURAghe [73] which exploit the cooperation, in modern Xilinx Zynq MPSoCs, between ARM cores and a convolution specific co-processor deployed in the reconfigurable logic. The co-processor has the peculiarity to embed a programmable soft-core releasing the ARM cores from most of the supervision duties and allowing the accelerator to be controlled by software at a fine granularity. This opens the way for an evaluation of both the ease of use of such a pre-designed template and its flexibility towards different devices allowed by the configuration of some architectural parameters. This evaluation has been completed in 2020 with the publication in [74].

Meanwhile, Xilinx released its proprietary FPGA-based acceleration engine called Deep Learning Processing Unit (DPU) [5]. It is probably the most powerful currently available DNN accelerator for FPGA. It provides an integrated framework, called Vitis AI [6], that helps designers in mapping CNNs on a templated soft IP. The DPU provides impressive performance on CNNs, using quantization and

high clock frequency in DSP slices. Quantization is required and can be applied automatically using a dedicated tool included in VitisAI. DSP slices are clocked at a very high frequency, using a *DSP Double Data Rate (DDR) technique* [7], which uses a 2x frequency domain to increase peak performance. Despite this, in some cases, the support for different network topologies is limited. For example, the support for TCN is affected by the lack of arbitrary dilation and kernel size support and the Vitis AI quantization process does not support 1D convolutions. Moreover, although the DPU offers the support for the Depthwise Separable convolution, it suffers from an inherent inefficiency given by the fact that the computational pattern is the same as in regular CNNs thus halving the exploitable parallelism.

For these reasons, subsequent phases of this work focused on these aspects. To the best of our knowledge, there were no published FPGA-based accelerators tuned to speed-up inference for generic TCNs at the time of the work presented in Chapter 3 which led to the published paper [16].

Lightweight Convolutional Neural Networks accelerators

The need to make CNNs more lightweight in terms of trainable parameters (memory footprint) and computational burden leads to the development of the so-called lightweight convolutional neural networks (LW-CNNs) such as MobileNet [48], ShuffleNet [122], SqueezeNet [51]. They have emerged to enable fast inference on embedded and mobile systems. However, the introduction of different operators like the Depthwise Separable convolution, calls for more specific accelerators. In literature, there is a non-negligible number of FPGA-based accelerators that specifically tackled this issue.

The work of Srivastava et al. [93] proposes an approach to depthwise separable convolution by implementing an FPGA based compute engine for this kind of operator in CNN. It is demonstrated for a single layer only and is intended to be a starting point in exploring mobile network components in FPGA.

Jiang et al. [55] presented a more structured architecture based on custom multiplexed processing elements supporting two dataflow modes in order to customize traditional convolution and depthwise separable convolution dataflow. It is implemented on a Xilinx XC7Z020 and tested on a simplified network architecture with the CIFAR10 dataset reaching up to 100 fps in the classification task.

The work of Liao et al. [69] has shown an implementation of the MobileNet model on the Xilinx XC7Z045 platform, designing a parallel acceleration scheme to increase the frame rate, and minimize the resource and power consumption in the form of multiplexing and configurability design. The maximum frame rate of this design is 5.52fps.

Bai et al. [12] proposed a scalable depthwise separable convolution CNN ac-

celerator designed to fit in FPGA of different sizes. It encloses all network layers into a Matrix Multiplication Engine (MME) Array unit composed of different processing elements. Despite its scalable nature, the sole implementation given on an Arria 10 SoC FPGA at 133 MHz leads to $266fps$ performance on a MobileNet execution.

Similar to this, in the work of Wu et al. [112] a multi-devices suitable architecture supporting depthwise separable convolution acceleration deployed in the Xilinx XCZU2EG and XCZU9EG MPSoCs has been presented. It is centred around two engines supporting different execution phases in parallel.

A slightly more complex approach deals with the reformulation and decomposition of the lightweight operations to reach a more efficient acceleration. It is the case of Light-OPU [118], an FPGA-based software programmable overlay processor with a compilation flow for general LW-CNN accelerations. It achieves 5.5x better latency and 3.0x higher power efficiency on average compared with edge GPU NVIDIA Jetson TX2.

Our approach tries to cope with the inherent inefficiency given by the underutilization of the FPGA processing capabilities, while dealing with the Depthwise Separable operator, by designing a dedicated accelerator that changes the traditional convolutional processing pattern, as will be described in Chapter 4.

Spiking Neural Network accelerators

The call for near-sensor efficient computation has brought not only a CNN model lightning trend but also to the diffusion of a new approach that relies on an event-based computation. This third generation of ANNs are called Spiking Neural Networks (SNNs) and are characterized by an internal neuron state, updated over time, which enables a data processing that begins only if the state surpasses a certain threshold. This event proportional data processing is ideal to reduce the application workload and to enhance energy efficiency by binding execution to events. As the execution in SNN is run-time dependent, exploiting classical hardware accelerators will result in efficiency drops.

The deployment of devices suitable to specifically accelerate SNN inference goes hand-in-hand with its growing interest, especially in applications at the edge [115]. Some solutions rely on custom hardware (ASIC) both for training and inference. TrueNorth [8] scalable chip consists of several cores implementing a function of synapses configured to map spikes to neurons. SpiNNaker [32], instead, is made up of small ARM processors featuring a custom interconnect communication scheme designed to be suitable for a large number of small spike-like messages. Loihi [25] is a many-core mesh comprising 128 neuromorphic cores with 1024 spiking neural units each communicating through an asynchronous Network on Chip (NoC). On the other hand, there is some work proposing FPGA-based

SNN accelerator designs. Bluehive [76] is a particular example of such a category. It is a large-scale real-time SNN simulation platform developed using Bluespec SystemVerilog consisting of 64 FPGA supporting up to 64k neurons implementing the complex Izhikevich model [53]. Minitaur [77] is an event-driven neural network accelerator implementing up to 65536 LIF neurons and 16.8 million synapses. It employs on-chip DSPs to carry out the fixed-point computation. It also maintains a hardware event queue that requires a sorting operation for each incoming event to support spikes with delays; this increases design complexity and run-time latency. S2N2 [61] is a streaming accelerator for spiking neural networks built upon FINN [98] platform, that relies on a custom method (streaming events instead of tick-batching) to reduce the memory utilization for inputs with a large temporal dimension. Gupta et al. [40] described an architecture for Simplified Spiking Neural Network which is implemented on FPGA and optimized for low power embedded applications with real-time learning, combining the neuron membrane model and a simplified on-line spike-time dependent plasticity (STDP [50]) learning. In the work of Han et al. [41] it has been proposed an FPGA-based SNN module implementation that relies on an enhanced hybrid updating algorithm resulting in a module that supports up to 16384 neurons. Also in the work of Ju et al. [56] it is possible to find a hardware architecture to enable the implementation of SNNs, with all layers in the network map on one chip, aiming to reduce latency by executing different steps in parallel. Fang et al. show a holistic optimization framework for encoder, model, and architecture design of FPGA based neuromorphic hardware [30]. They present a neural coding scheme and training algorithm to enable fast inference, a model in which SNNs are represented as networks of Infinite Impulse Response (IIR) filters and an end-to-end framework to optimize and deploy FPGA implementation.

Many of these works focus their attention on the implementation of Networks with Fully Connected layer while only a few of them rely on networks with a structure similar to the one found in the CNNs. However, the majority of them are fully integrated into the device.

Differently, we have tried to apply the NEURAghe model also to SNNs, so that it is possible to execute different network topologies without reconfiguring the device by exploiting the cooperation between ARM processors and the softcore in programmable logic to decompose the execution. Our work result of the collaboration with the Digital Circuits and Systems group of ETH Zurich starts from the adaptation of an ASIC-tailored Spiking Neural Engine to FPGA and sets the basis for the development of an FPGA-based SNN accelerator topology agnostic.

1.4 Reference Architecture: NEURAghe

NEURAghe [73] is a CNN inference accelerator architecture that exploits the cooperation between an ARM Processing System (PS) and the Programmable Logic (PL) in Xilinx Zynq devices. Communication at the PS-PL interface is allowed by two high-performance 64-bit ports to access the memory-mapped off-chip DDR, and two general-purpose 32-bit ports for memory-mapped control of the NEURAghe architecture and standard output. As can be seen in Figure 1.4,

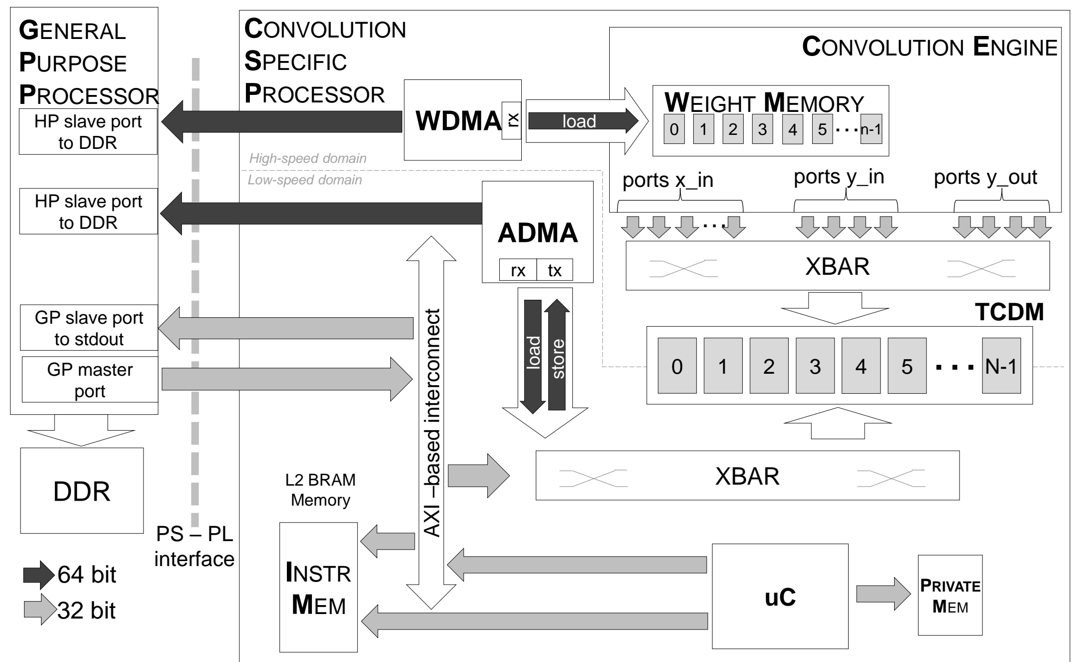


Figure 1.4: NEURAghe architecture

the Programmable Logic hosts a Convolution Specific Processor (CSP), dedicated to accelerating compute-bound tasks (convolution), while the processing system acts as a General Purpose Processor (GPP), in charge of executing tasks that are hard to accelerate. With such a template the accelerator can support the deployment of arbitrary CNN topologies by offloading the execution of its layers to the GPP or CSP according to their characteristics.

NEURAghe can be programmed by defining a network described in C language, which acts as a front-end application to be executed by the GPP, which can include calls to adequate APIs, outsourcing tasks to the CSP, or actual processing functions executed on the ARM processors. To enable this paradigm,

NEURAghe integrates a RISC-V[1] based lightweight processor inside the CSP. The RISC-V core runs a back-end firmware (middleware). Upon the receipt of a CSP-triggering command, issued by the GPP, the middleware schedules data transfers and convolutions implementing the requested operation, without PS intervention, leaving the latter available for actual computation workload. As an example, the referenced work of Meloni et al. [73] has shown how the ARM cores can be used to build a balanced software pipeline in cooperation with the CSP implemented on the PL.

1.4.1 Convolution Specific Processor

The Convolution Specific Processor is made by various System Verilog HDL described sub-modules. As said above, the soft-core manages operation scheduling inside the CSP and relies upon an instruction memory, loaded by the GPP and a private memory to be used during execution. There are two Direct Memory Access (DMA) Modules, one to transfer the weights inside the Weight Memory (WM) and the other to transfer activations both inside and outside the tightly-coupled data memory (TCDM). This is the main CSP memory in charge to store input and output data and partial computation results. It is composed of several independently accessible banks, made up of the dual-port BRAM slices of the FPGA partitioned in two sections and connected both to the ADMA and the Convolutional Engine (CE) that can access at the same time. TCDM banks dual-port nature and its partitioning allow the implementation of a double-buffering technique. Indeed, when the CE is reading inputs for the actual computation phase from one of the sections of the banks, ADMA could transfer those for the next phase. In a similar way, when the CE is writing actual output results, ADMA could transfer to the DDR those from the previous computational phase.

Convolution Engine

Figure 1.5 shows the CE configuration. It is the main CSP module and the computational core of the architecture dedicated to accelerate Multiply and Accumulate (MAC) operations execution, it is composed of a MAC Matrix of N_{cols} columns by N_{rows} rows of Sum of Product (SoP) units in charge of calculate the contribution of $3 \times N_{cols}$ or N_{cols} input features (IFs) (depending on $kernel_size$ dimension) to N_{rows} output features (OFs). N_{rows} Shift Adder modules sum together partial results from SoPs in each row with data values resulting from the N_{cols} previously computed input feature partial results that are read from on-chip memory, enabling successive accumulation over multiple CE runs. The architecture is designed to

[1]RISC-V is an open standard reduced instruction set architecture provided with open source licenses.

process convolutions with a 16-bit fixed-point data format without restrictions on the number of fractional bits in the specific layer configuration. Input feature data are fed through a set of line buffers modules, incorporated in the Activation source module of the Figure 1.5, that by caching values from the input feature lines are capable to load an entire image window to be convoluted, at every new input pixel, with weight filters. In particular, each SoP receives two square convolution windows per cycle performing MAC operation by applying to them the bi-dimensional kernels loaded by the Weights source through an inner Weight Loader module.

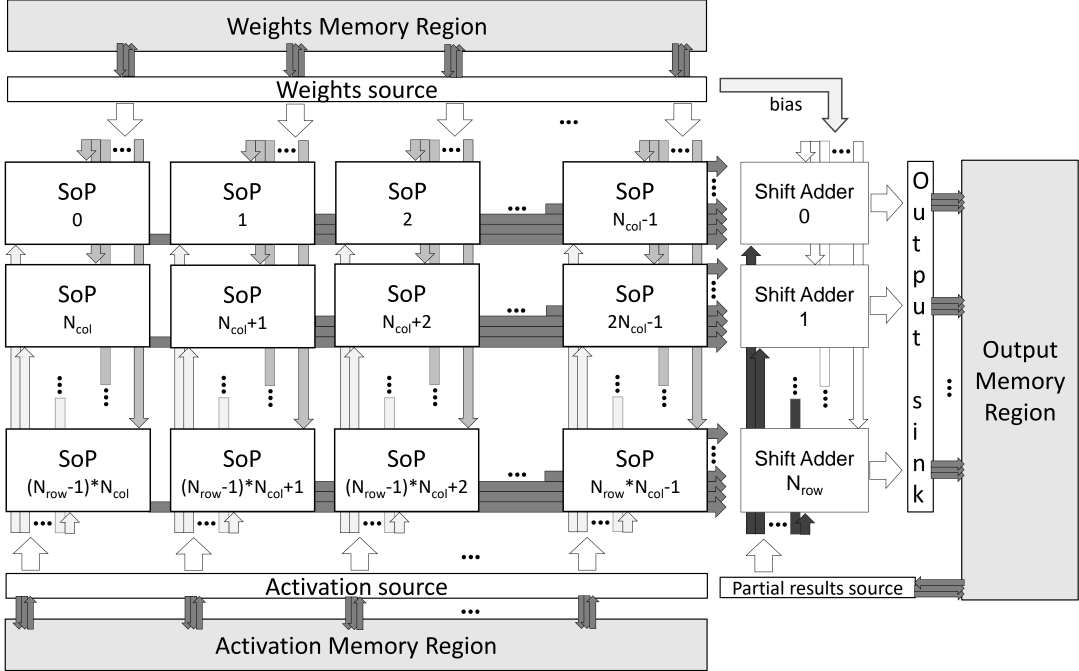


Figure 1.5: Convolution Engine. $N_{rows} \times N_{cols}$ MAC Matrix

By looking at the Figure 1.4, the Weight Memory Region is the one containing the WM, while Activation Memory Region and Output Memory Region are composed by TCDM banks which are in charge of storing activation and partial/output results respectively (depicted as in figure for ease of representation).

Line Buffers

Each SoP in the CE is fed with a Line Buffer whose internal structure is depicted in Figure 1.6.

LB blocks are realized by means of 32-bit wide shift registers thus, after an initial preloading phase they are capable of reading 2 pixels per cycle and feeding SoPs with two square convolution windows per cycle. Line Buffers are capable of

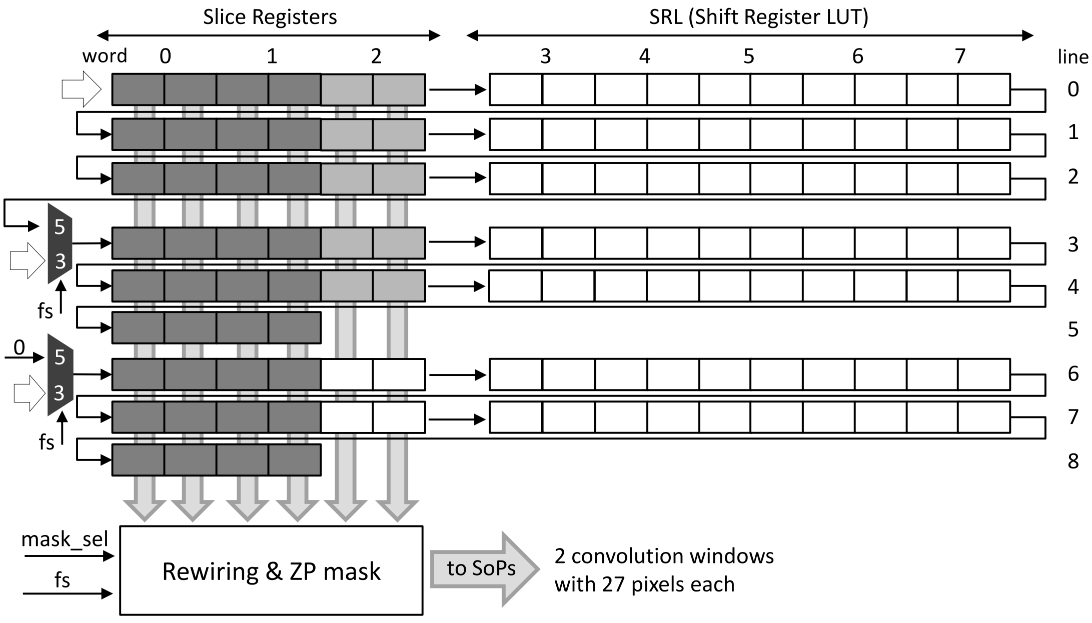


Figure 1.6: Linebuffer structure

adapting at run-time to two different kernel sizes, 5×5 and 3×3[2], by means of some selection circuitry, which is configured by the middleware through memory-mapped registers accessible by the soft-core. In particular, with a fixed number of Slice Registers (dark and light grey coloured in Figure 1.6) and the necessary glue logic, LBs are capable of supplying SoP with up to two sets of 27 pixels per cycle, as they contain two independent sets of pipelined and cascaded DSPs, as will be described in the next section. The 27 pixels can be used as three independent sets of 3×3 convolution windows from three different input features or as a single 5×5 convolution window from a unique input feature (leaving two LB registers and SoP DSP unused in this case). Additional logic is used to apply zero-padding when needed and to allow the right LB-to-SoP connection.

SoP modules

Each Sum of Products unit in the MAC Matrix are pipelined with a structure made up of trellises of multiply and add operations (a multiplier, an adder and two pipeline registers), as shown in Figure 1.7, to maximize mapping efficiency on the

[2]These are the only two natively supported kernel sizes by this version of the CE (the most adopted by SoA networks at the time of the architecture design). Different sizes could be handled by means of the GPP cooperation [73]. This drawback, on the other hand, motivated the improvements carried out in subsequent versions and described in the following chapters

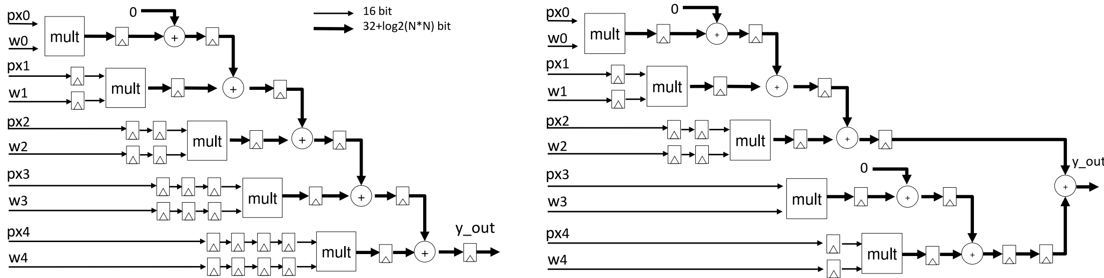


Figure 1.7: Single and Multi-trellis SoP cascade

FPGA DSP resources. To deal with the throughput of two convolution windows per cycle produced by the LBs, each SoP module includes two sets of parallel trellises, for a total of $2 \times N^2$ DSP blocks (where N is the size of the 2D kernel). The multi-trellis disposition in the right side of Figure 1.7 is an architectural choice made with the purpose to avoid restrictive placement constraints in the FPGA place & route phase.

1.4.2 Execution scheduling

The RISC-V soft-core in the CSP runs a Middleware coded in C, loaded and activated at the startup of the system. Once received the GPP programming commands with convolutional layer characteristics and the starting trigger, the core partitions layer executions in sub-parts.

First of all, it activates DMAs to load in a bank section of the WM and TCDM a number of input features and kernels such that it is possible to activate the entire MAC Matrix and thus by feeding it with $3 \times N_{cols}$ or N_{cols} IFs, depending on the kernel size (3 or 5 respectively) and with a number of kernels that is related to the number of IFs and OFs which are to be produced in a single matrix activation (multiple of the N_{rows} parameter).

The baseline for the MAC Matrix full execution, except for corner cases[3], is what outlined below:

$$IF = 3 \times N_{cols},\ kernels_{3\times3} = 3 \times N_{cols} \times N_{rows} \rightarrow OF = N_{rows}$$

$$IF = N_{cols},\ kernels_{5\times5} = N_{cols} \times N_{rows} \rightarrow OF = N_{rows}$$

which are the contribution of $3 \times N_{cols}$ or N_{cols} IFs to N_{rows} OFs by applying $3 \times N_{cols} \times N_{rows}$ or $N_{cols} \times N_{rows}$ kernels.

[3]A corner case is when the multiplicity of one of the layer parameters (IF or OF) is not an integer multiple of one of the MAC Matrix parameter (N_{cols} or N_{rows}) for which there will be an execution phase when the matrix will not be fully active.

However, to exploit data reuse, when allowed by the architectural configuration, IFs are kept in memory to let the MAC Matrix calculate their contribution to a number of OFs multiple of N_{rows} namely an output group (OG). This requires also loading a number of kernels that grows proportionally with OG. After the first loading phase, the Middleware triggers the execution phase, activating the MAC MAtrix. Meanwhile, it programs DMAs to load the data for the next execution phase in the other bank section both of the WM and the TCDM (double-buffering), ready to be used in the forthcoming execution phase.

Once the contribution of all IFs to the actual OG has been calculated, the Middleware schedules a store operation where the DMA transfers OFs from the TCDM to the DDR.

A scheme of the alternation of these phases is depicted in Figure 1.8.

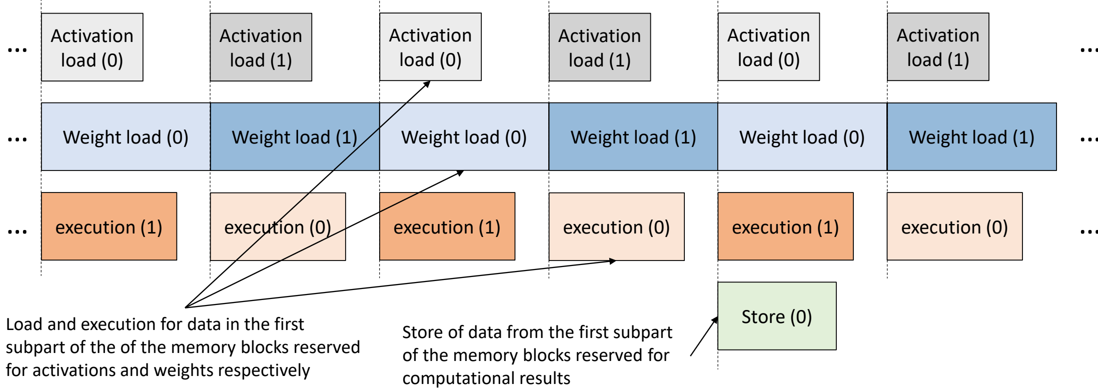


Figure 1.8: Scheduling scheme

In addition to the double-buffering technique, to keep the execution pipeline as balanced as possible, according to the layer characteristics, it is possible to choose an appropriate value of OG to let execution phases cover as much as possible transfer phases.

2 | Exploring device parameterisation

The wide scope of use-cases that can be found when dealing with AI-related problems at the edge ranges from low power and low-cost devices to more resource consuming ones. In this chapter, starting from the NEURAghe accelerator, described in Section 1.4, we explore the capabilities of this architectural template to scale at design-time to different MPSoCs with different costs, sizes and resources availabilities by leveraging on its tunable parameters like the number of CSPs, the MAC Matrix size and the data precision. We test different solutions over a range of target boards demonstrating that this architectural template is able to support different trade-off optimization scenarios.

2.1 Parameter exploration

Figure 2.1 shows the architectural template while it is configured to host multiple Convolution Specific Processors (CSPs). Its number can grow according to the target board capabilities in terms of resources depending on the PS-PL communication port. They can be used in parallel to map independent layers. In this case, CSPs can be considered as independent IPs to be instantiated at design time in a relatively easy way, by properly connecting PS-PL ports. Their activation is managed by the GPP.

As depicted in Figure 1.5 the MAC Matrix is composed of a parametrizable number of SoP Units that can be chosen (at design time) acting to the N_{cols} and N_{rows} parameters, according to the target board characteristics, namely, for example, available DSP slices. Furthermore, it is worth considering that the number of SoPs is directly related to the input and output features processed.

Another selectable parameter is data precision. It can be 16-bit (as in the original version) or 8-bit (introduced ad-hoc) and it can be chosen at design or run-time. In any case, all data will have the same bit width representation and the trade-off will be between performance and accuracy.

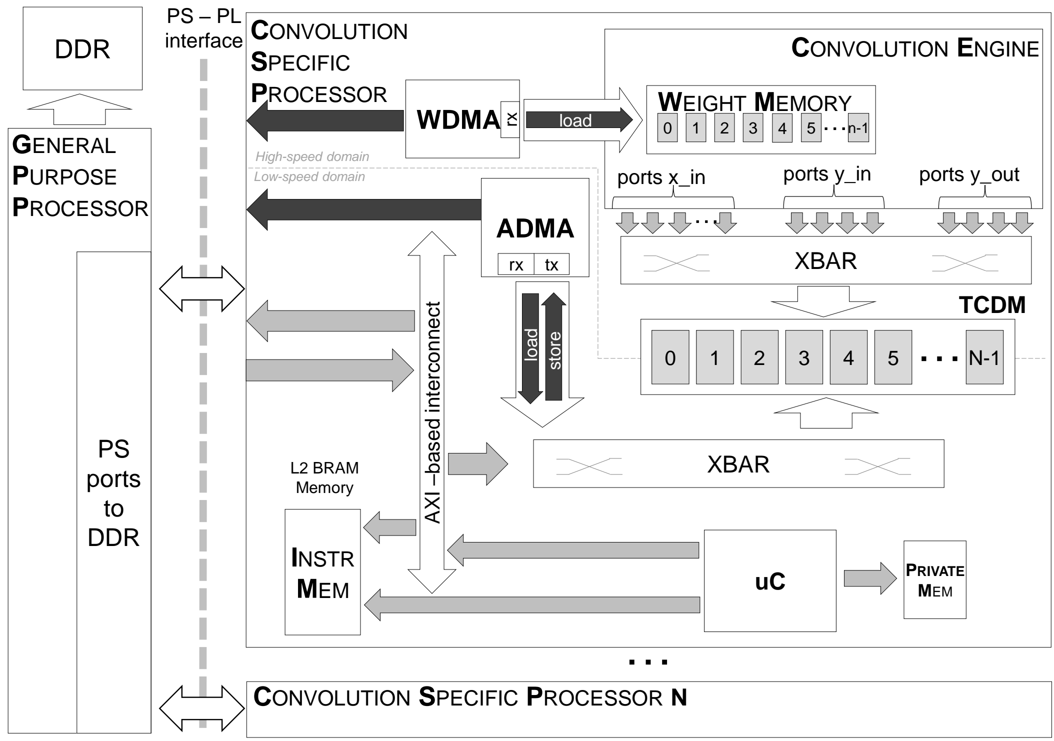


Figure 2.1: NEURAghe architectural template with multiple CPSs

8-bit support

To support 8-bit data precision we introduce a low-impact modification to as few as possible components of the CSP, resulting in architecture whose congestion level is comparable with the original version.

The main change concerns DSP configuration and the resulting changes to the datapath of SoP modules. Following the technique depicted in the Xilinx white paper of Fu et al. [31], we enable DSPs to exploit its internal pre-adder module by summing together 8-bit input pixels, a and d, belonging from two adjacent convolutional windows, by means of the two 25-bit wide pre-adder input registers. One of the two pixels is 17-bit left-shifted while the other is sign-extended such that the pre-adder output register preserves both input values in its Least Significant Bit (LSB) and Most Significant Bit (MSB) section respectively, resulting in $a \cdot 2^{17} + d$. This value feeds the internal multiplier together with the sign-extended weight value b, and again, preserving the two independent multiplication results in the 45-bit wide multiplier output register[1], $a \cdot 2^{17} \cdot b$ and $d \cdot b$, in the MSB and LSB

[1]For the MSB part this is true only if $d \cdot b$ is non-negative, otherwise it corresponds to $a \cdot b - 1$, which is equivalent to sum the sign bit.

parts respectively.

The DSP surrounding circuitry into SoP modules has been slightly modified to allow the proper data propagation to the DSP input registers and to ensure support for both data precision. The same happens for LBs and Shift Adders modules. The data precision selection mechanism depends on a memory-mapped value which is defined by the GPP during its CSP programming phase and is propagated through the aforementioned modules.

2.2 Experimental results

Given this range of choices, aiming to provide a broad scope of scenarios, the configurations that have been evaluated are those in Table 2.1. For each one we report the target SoC, the number of CSPs instantiated, the MAC Matrix dimension and the resource utilization for the relevant FPGA slices. All reference devices belong to the Xilinx Zynq-7000 family but differ in size, cost, power consumption and resource utilization.

Table 2.1: Resource occupation on different configurations

Target Soc	XC7Z045	XC7Z045	XC7Z020	XC7Z007S
CSP num	1	2	1	1
SoPs	4×4	2×4	2×2	1×1
DSP	864 (96%)	864 (96%)	216 (98%)	54 (82%)
LUTs	99546 (51%)	101352 (46%)	43880 (83%)	12610 (87%)
BRAMs	320 (59%)	288 (53%)	136 (97%)	44 (88%)

The bigger XC7Z045 SoC is capable of hosting two configurations with different MAC Matrix shapes and numbers of CSP, while XC7Z020 and XC7Z007S host decreasing matrices sizes. Each configuration has been evaluated for both available data precisions.

MAC Matrix Size

Table 2.2 reports a performance evaluation for different MAC Matrix configurations over different Xilinx Zynq SoCs operating on ResNet-18, VGG-16 and SqueezeNet target networks. The choice of the Matrix size could be done willing to optimally exploit the DSP slice usage of the target device. As it can be seen, implementations over bigger SoCs lead to higher absolute performances, while smaller ones benefit from higher efficiency with respect to the peak achievable by the architecture, as depicted in the roofline model in Figure 2.2.

For smaller configurations the operational intensity required to approach the peak is limited by the available resources, thus it is easier to be reached while processing the majority of the convolution layers. On the other hand, larger matrices require a greater number of operations to be performed to reach the maximum level of performance without being restricted by the PS-PL bandwidth. This can be seen in Figure 2.3 showing the efficiency trend for some of the different configurations with respect to ResNet-18 and VGG-16 convolutional layers: an increase in layer operational intensity has a direct consequence on the achievable efficiency. General lower performance for ResNet-18 layers is due to their particular characteristics requiring often the GPP intervention.

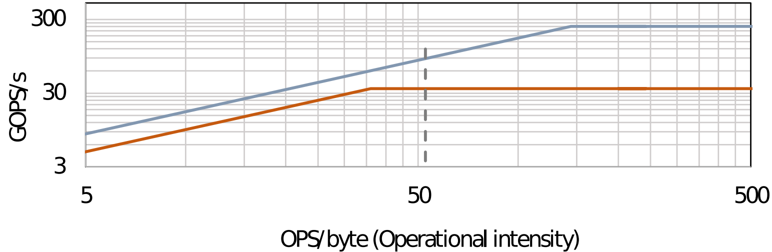


Figure 2.2: Roofline model for the 4×4 (blue) and 2×2 (red) configurations. The vertical dashed line represents the average operational intensity of a convolutional layer.

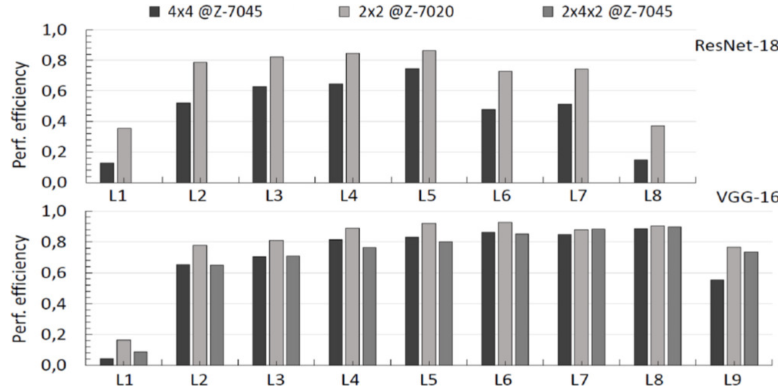


Figure 2.3: Comparison of the performance efficiency, achieved on all the convolution layers (L) in ResNet-18 and VGG-16, by different configurations, estimated as the ratio between actual performance (GOps/s) and peak performance (GOps/s). Efficiency for the double cluster configuration on ResNet-18 is not reported as it is mainly limited by the GPP

Multiple CSPs

In order to exploit advantages of bigger devices with those of smaller matrix size, we evaluate a particular configuration by instantiating multiple CSPs with a 2×4 matrix. Characteristics are shown in Table 2.1 and performance has been evaluated over ResNet-18 and VGG-16. As can be seen, the overall performance is better for regular layers like VGG ones as they do not need the relevant intervention of the GPP for accessory operations like data marshalling.

Table 2.2: Performance evaluation for different MAC Matrix configurations

MAC Matrix	4×4	2×2	4×4	2×2	1×1
SoC (Price)	XC7Z045 ($2500)	XC7Z020 ($450)	XC7Z045 ($2500)	XC7Z020 ($450)	XC7Z007S ($89)
Freq [MHz]	140	120	140	120	80
Benchmark net	ResNet-18	ResNet-18	VGG-16	VGG-16	SqueezeNet
GOps/s (16-bit)	61.9	16.1	172.67	42.5	7.46
GOps/s/W (16-bit)	6.19	4.6	17.26	12.56	2.98
GOps/s/k$ (16-bit)	25	37.78	69	97.5	95
GOps/s (8-bit)	111.12	29.04	335.09	84.77	14.05

Table 2.3: Performance evaluation for multiple CSPs

MAC Matrix	dual 2×4	dual 2×4
SoC (Price)	XC7Z045 ($2500)	XC7Z045 ($2500)
Freq [MHz]	140	140
Benchmark net	ResNet-18	VGG-16
GOps/s (16-bit)	59.8	188
GOps/s/W (16-bit)	5.98	18.8
GOps/s/k$ (16-bit)	21.4	75.2
GOps/s (8-bit)	63.9	370.4

Arithmetic Data Precision

Both Table 2.2 and 2.3 show performance trends for each configuration with respect to 16-bit and 8-bit data precision. As can be seen the increment in performance is almost of a factor 2 for the latter choice. This is because, in this case, exploiting the DSP capabilities it is possible to map 2 MAC operations which are executed together at every cycle [31].

2.3 Comparison and summary

In summary, architectural parameterisation leads to different configurations that can be used in different contexts to fit in different SoCs ranging from more expensive but resourceful to cheaper ones. In the literature is not easy to find the same degree of design-time flexibility and runtime selection of data precision. Table 2.4 shows a comparison with some works with respect to the execution on VGG-16. In some cases, the performance results are comparable but this architecture delivers more flexibility while in other cases also better results are achieved. Furthermore, the implementation of an extremely low-cost device is a rarity in the landscape of FPGA-based hardware accelerator for classic CNN inference.

Table 2.4: Comparison with alternatives in literature

	NEURAghe	[101]	[88]	[38]	**NEURAghe**	[103]	[38]
SoC	**XC7Z020**	XC7Z020	XC7Z020	XC7Z020	**XC7Z045**	XC7Z045	XC7Z045
Freq[MHz]	**120**	125	150	214	**140**	125	150
GOps/s (16bit)	**42.5**	48.53	31.38	-	**172.67**	155.81	137
GOps/s (8bit)	**84.77**	-	-	84.3	**335.09**	-	292

3 | TCN inference optimization on FPGA-based accelerator

Convolutional Neural Networks have become popular for computer vision applications, such as image recognition [43, 65, 113], object detection [96] or video frame classification [59], which are inherently bi-dimensional applications. However, convolutional networks have also been extended to deal with time sequences (sequence modelling task). These CNN variations are usually indicated as Temporal Convolutional Networks (TCNs) [66]. Multiple TCN architectures have been proposed, reaching impressive performance on edge-related tasks such as sentence classification [63], speech recognition [42], text understanding [111], Natural Language Processing tasks [85] and, more recently, machine translation [58], audio synthesis [99], language modeling [24] or signal sequence analysis in the healthcare domain, as action detection [62] or ECG classification [35]. Research work of Bai et al. [13] demonstrates that the exploitation of a Temporal Convolutional Network for typical sequence modelling tasks can convincingly outperform older and better-known Recurrent Neural Networks [46] [18]. In particular, they leverage a residual block unit implementing a dilated causal convolution with different dilation factors as the baseline for their network architectures. The dilation factor is a key concept in TCNs as it enables an output at the top level to represent a wider range of inputs, thus effectively expanding the receptive field (defined in the following) of the network without increasing its depth and thus its parameters.
Thus, the rapidly increasing interest in TCNs pushes for investigating on acceleration for these networks. Especially in the embedded domain, where (near) real-time analysis of sequences of data samples acquired by sensors is a common case, accelerating this kind of workload on reconfigurable devices is a very appealing approach. While FPGA based inference accelerators for classic CNNs are widespread, the literature is lacking in a quantitative evaluation of their usability on inference for TCN models.

In the following, this chapter will show:

- An exploration of the capabilities of a state-of-the-art CNN inference accelerator [73] specifically enriched to provide the flexibility needed in TCNs, to

support freely selectable *kernel sizes* and *dilated* convolutions, with freely selectable *dilation rates* and *stride values*;

- A performance evaluation (the first to the best of our knowledge) over various benchmark TCNs, focusing on implementation on low-cost and low-power all-programmable SoCs, more suitable for the integration of edge-computing and IoT processing nodes, considering two widely accessible devices in the Zynq and the Zynq Ultrascale+ families;
- A methodology for the optimal execution/scheduling of data-transfers exploiting the specific sequence-based structure of data in TCNs;
- A methodology for improving efficiency based on *sample batching* (sequence buffering);

Results are reported on absolute execution time as well as on the efficiency of the execution with respect to the peak performance imposed by the device resources. Finally, the efficiency of the FPGA-based acceleration will be assessed, comparing with software execution and with state of the art accelerators on bi-dimensional CNNs.

3.1 TCN model generalities

Time sequence modelling with artificial neural networks is historically associated with Recurrent Neural Networks (RNNs) [28][108][36]. These types of networks are dedicated sequence models that maintain a vector of hidden activations that are propagated through the time where its state acts as a representation of what has been seen so far in the sequence. Their memory capabilities leverage feedback loops. Thanks to decades of research efforts focusing on their improvement and their applications, such networks are extremely popular and used with huge success in language modelling [94][37][45] and machine translation [95][10]. However, there is common knowledge about difficulties associated with the training of RNNs [14][79], partially moderated by the introduction of Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs). Their aim is to resolve RNN's training vanishing gradient problem introduced by feedback loops by simplifying the network structure (GRU) or using memory cells in addition to standard units (LSTM) [21]. In the end, however, they still exhibit an inherent complexity. This has driven the research of alternative solutions to RNN, often focusing on convolution-based operators. A very promising alternative approach is based on TCNs. In [13], authors propose a thorough comparison between TCNs and recurrent architectures on benchmarks commonly used to assess RNN variants. Although highlighting some potential disadvantages of TCNs, such as requiring more memory than RNN during inference and being less prone to the application of transfer learning techniques, authors define several key advantages that can be exploited during network design and training, such as, for example, inherent parallelism to be exploited to speed-up computation, adaptability to different domains requiring different memory requirements, and stability of the gradient during training.

3.1.1 Mono-dimensional dilated convolution

A dilated convolution operation F on element s of a sequence [13] can be defined as:

$$F(s) = (x *_d f)(s) = \sum_{i=0}^{k-1} f(i) \cdot x_{s-d \cdot i} \quad (3.1)$$

where $x \in \mathbf{R}^n$ is a 1-D input sequence, $f : \{0, ..., k-1\} \in \mathbf{R}$ is a kernel of size k and d is the *dilation rate*.

The sequence of samples that constitute the input of a TCN can be processed both off-line or in real-time streaming. The second case is very useful in application cases requiring continuous analysis of the input sequence, e.g. aimed at

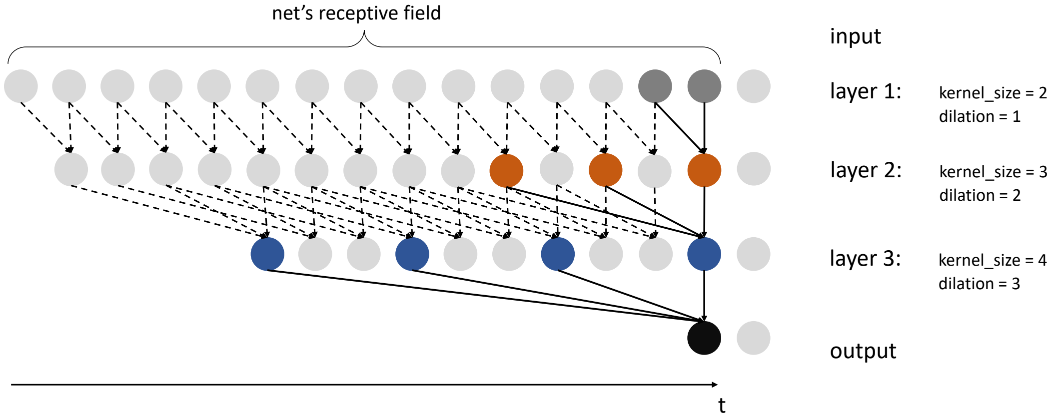


Figure 3.1: TCN execution graph. For this example, see (3.2), $receptive field = 1 + (2 - 1) \times 1 + (3 - 1) \times 2 + (4 - 1) \times 3 = 15$

the identification of specific events and/or at promptly closing the loop on data-triggered actuations. This implies that the sequence must be analyzed at every time step, after being updated with a new sample. It is possible to identify the minimal sequence size to produce a valuable output sample as the *receptive field* that depends on convolutional layer parameters such as the *kernel_size* and the *dilation rate*:

$$receptive field = 1 + \sum_{l=1}^{L} [k(l) - 1] \times d(l) \tag{3.2}$$

where $l \in 1, 2...L$ is a layer of the network. This can be thought of as a sliding processing window for the input sequence.

Figure 3.1 highlights and generalizes these concepts. In particular, for the fictitious network shown, dilation rates increase linearly through the depth of the network, although it is common practice to use an exponentially growing value [13],[66],[99]. This is probably a property inherited from signal processing elaboration [47],[27],[90], but it can be generalized [116]. It is worth noticing that increasing the dilation parameter leads to an increase of the network's memory without affecting its depth.

3.2 TCN supporting hardware features

In order to support TCN execution, the NEURAghe CPS IP has been enhanced with new features to improve flexibility. This required substantial modification of the Convolution Engine (CE), and an improvement of the circuitry and procedures managing transfers to/from the DDR memory.

As seen in Section 1.4, in its original version, each MAC module inside the CE was composed of a multi-trellis cascade structure of DSP slices, optimized to execute convolutions featuring a fixed range of kernel shapes. Different shapes and sizes of the kernel had to be implemented reusing the MAC module over time to compose bigger kernels or underutilizing it to implement smaller ones, resulting in significant performance loss. Such an organization fits well with CNNs, where kernel sizes are usually square and selected among a quite limited number of options. In TCNs, the variability is higher. Thus, MAC modules have to be re-designed to be more flexible, as described in more detail in Section 3.2.1. Moreover, to efficiently deal with bi-dimensional windows of pixels, as it is common in CNNs, pixel loading in [73] was implemented using a *line buffer* at each input port of the CE. In this way, several lines of pixels are stored in the buffer and every pixel loaded by the accelerator enables a new convolution at every cycle. This, unfortunately, makes the implementation of dilation and strides complicated, thus it is not a good practice for TCNs, that require very often highly variable strides and dilation rates. Thus, we have modified the memory access circuitry, depriving activation source of line buffers and adding flexible and programmable memory management modules, that are described in more detail in Section 3.2.2. In order

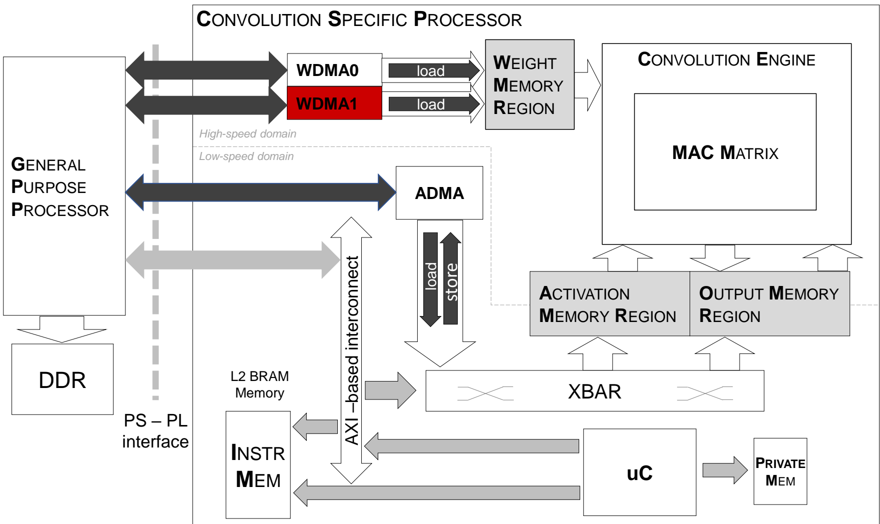


Figure 3.2: NEURAghe architecture with double weight DMA

to increase the bandwidth to external memory, we have also added a second Weight DMA (WDMA) unit. The two WDMAs work simultaneously sharing the load of weight transfer needed by each computational phase. Finally, we have adapted the scheduling of DMA transfers and convolutions, implemented in the previously mentioned middleware, to the characteristics of TCNs, considering each layer's receptive field to determine the number of samples and weights that have to be transferred for every CE activation, as described in more detail in Section 3.2.3 and 3.2.4. Considering what said above, the CE has to be designed according to the following accelerator principles:

- It has to be $kernel_size$ agnostic;
- It must execute convolutions with multiple stride values without performance overhead;
- It must support freely selectable dilation values.

3.2.1 Freely selectable kernel sizes

To support arbitrary kernel sizes, every DSP cell is dedicated to an entire convolution kernel computation, reusing it over a number of cycles depending on the kernel size. A new sample of the output feature under production is thus produced after $kernel_size$ cycles and is ready to be sent to the Shift Adder module. Using one single DSP cell per kernel can easily require the instantiation of a very high number of SoPs, with the aim of exploiting as many resources as possible among those available on the target device. To keep the MAC Matrix growth feasible, SoPs are designed to be composed of 4 Xilinx DSP48E primitives, performing 4 $MACs/cycle$, operating in parallel on 4 different 16-bit samples, as visualized in Figure 3.3, from 4 neighbour convolution windows in an input feature.

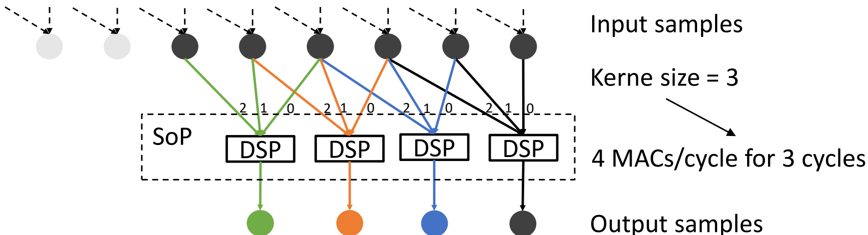
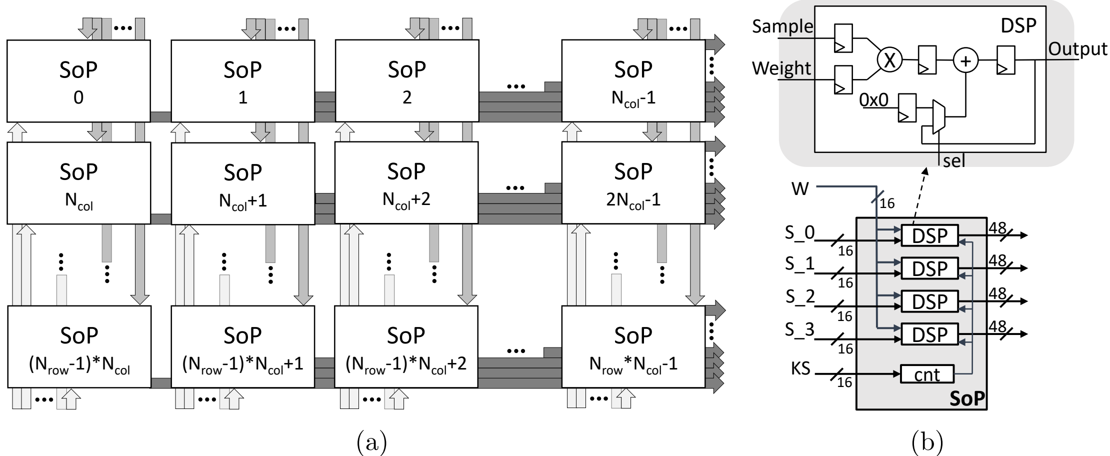


Figure 3.3: SoP elaboration scheme

Figure 3.4b represents the organization of a SoP. Considering the template in Figure 3.4a (already presented in Section 1.4), the MAC Matrix implementation

requires a deterministic number of DSP48E primitives, as indicated by Equation 3.3.

$$N_{DSPs} = N_{rows} \times N_{cols} \times 4 \tag{3.3}$$



(a) (b)

Figure 3.4: MAC Matrix (a) and Sum of Product Unit (b)

3.2.2 Flexible activations and weights fetching

To enable arbitrary stride and dilation values, the fetching of input samples from the internal memory has been designed to be very flexible. Figure 3.5 shows such an organization. Each CE port dedicated to input samples is endowed with a programmable Activation Source module, while weight kernels are fetched from Weight Memory banks by means of Weights Source modules. Such source modules can be programmed at start-up according to stride, dilation, aspect ratio and size of activations and weight kernels. Fetching of bi-dimensional memory sections is enabled, to support generic CNN execution.
The memory subsystem has been designed to enable conflict-less loading of neighbour convolution windows, in the following manner. The Activation Source module controls N_{cols} ports of the convolution engine. Each one loads samples from a dedicated Block-RAM (BRAM) module. Considering that four samples, belonging to different windows, can be loaded in the same cycle, to support stride values up to 3 samples, each module of the activation memory has to be composed of at least 8 different independently accessible RAMB18 modules. Figure 3.6 represents an example where a different configuration of RAMB18 modules can determine a conflict.

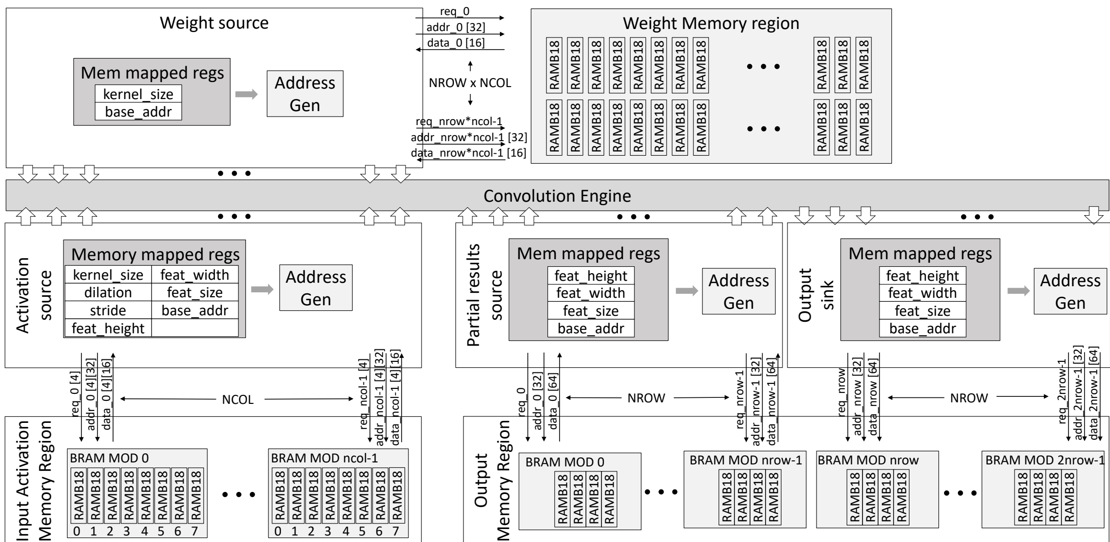


Figure 3.5: CE memory transfers configuration. Communication between memory regions and CE is made by means of configurable source modules which have access to BRAM modules through the exposed B-ports. Each source module can be programmed according to layer characteristics and it is able to feed each SoP Unit with 4 input samples/cycle from the Activation Memory Region and weight kernel elements from Weight Memory Region, both composed by independently accessible memory banks. Moreover, with the same behaviour, a source and a sink module handle transfers from Output Memory Region by feeding Shift Adder Modules with partially computed values from the previous step and storing actual results.

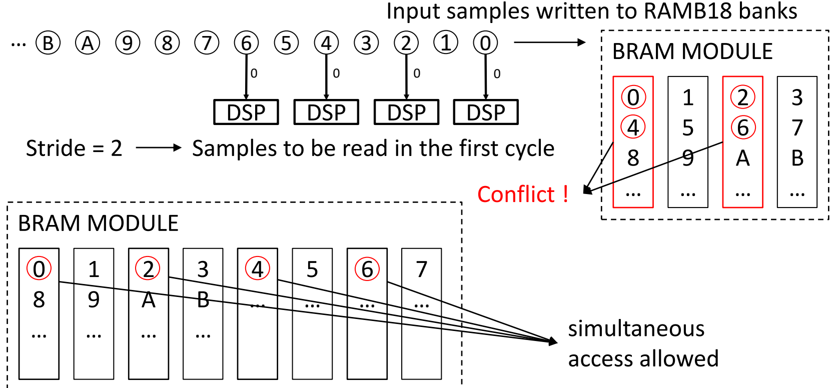


Figure 3.6: BRAM read conflict example. Samples are stored in BRAM modules using interleaving. Consecutive samples are stored in adjacent RAMB18 banks. 4 DSP slices in a SoP unit, with stride 2, in the first cycle of the convolution, load respectively samples 0,2,4,6. With four RAMB18 banks, samples 0-4 and 2-6 are on the same bank, creating a conflict. The bottom part shows how the conflict is avoided by doubling the number of banks.

A stride value of 3 is sufficient to support most of the TCN use-cases available in literature. However, following the same method, this support could be further extended. But that would have meant reconsidering the overall bank slices distribution depriving other sections of the design thus resulting in a smaller MAC Matrix size and therefore a poor DSP utilization. Considering that a stride value wider than 3 is very unlikely to be used in convolutional layers, we choose the aforementioned configuration. Moreover, the Weight source module controls one port per SoP in the matrix, which has to be implemented by at least one RAMB18 module.

Finally, the CE has a set of ports that are used to write results and to load previously computed partial results, when a convolution requires to accumulate over several accelerator operations. These ports are controlled by a Partial Result Source module and by an Output Sink module. Each of these modules controls N_{rows} ports, each one writing/reading four samples simultaneously. Thus BRAM modules in the corresponding memory region are composed of at least 8 RAMB18 modules each.

Samples and weights, in the experiments presented in this work, are all using a 16-bit data format, thus RAMB18 modules are configured to expose two 16-bit addressable ports and can be 1024 words deep.

Considering the described organization, a given architectural configuration requires a number of RAMB18 primitives that can be deterministically estimated as indi-

cated in Equation 3.4.

$$N_{BRAMs} = N_{rows} \times N_{cols} + N_{cols} \times 8 + (N_{rows} \times 2) \times 8 + 32 \quad (3.4)$$

The first component corresponds to weight memory, the second to the modules storing input activations, the third to output and partial results memories. 32 blocks are used to implement the RISC-V scheduler instruction memory and private memory.

3.2.3 TCN support in firmware

As depicted in Section 1.4, from the software point of view, the execution model in NEURAghe leverages the cooperation between the ARM-based processing system and the RISC-V soft-core implemented in the programmable logic. When the CSP has to be used, the program in the PS sends commands describing the layer to be executed. The RISC-V soft-core decodes the command and decomposes the layer in sub-operations, namely partial convolutions in the accelerator and data transfers from/to the off-chip memory, executing an optimized firmware which is also coded in C. The firmware uses a double-buffering technique, to allow the accelerator to overlap transfers phases with convolutions, reducing as much as possible idle times in the CE to exploit the processing capabilities of DSP slices with maximum efficiency.

When considering TCN executions, the previously described paradigm has to be applied to the characteristics of the algorithm. For every Convolutional Layer in a TCN, given its *kernel_size* and a *dilation*, it is possible to consider a local *receptive field*, indicated as RF_{local} in Equation 3.5, that is the minimum amount of layer's input samples per channel needed to produce a valuable output sample.

$$RF_{local} = 1 + (k - 1) \times d \quad (3.5)$$

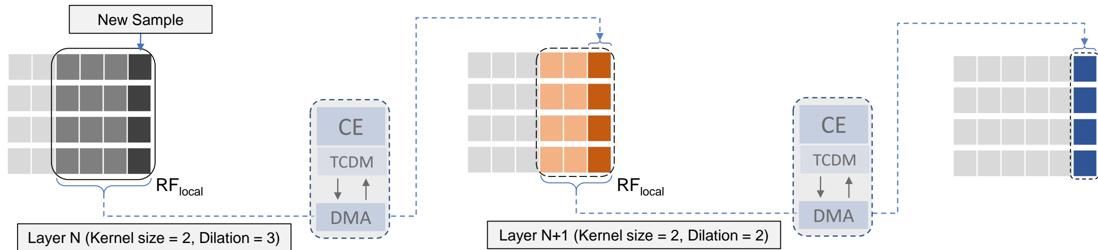


Figure 3.7: TCN execution on NEURAghe: *batch size* = 1

Figure 3.7 shows how input and output transfers are implemented for TCNs. At every new time step, the input to a layer is updated by adding one new sample

to the input features. In order to execute the layer, the firmware triggers Direct Memory Access (DMA) transfers to load RF_{local} samples per each input feature to the activation input memory. After the execution, one output sample is produced per output feature. Output samples are sent to DDR using an output DMA transfer, to be stored until the next time step.

3.2.4 Improving through batch processing

Although the previous approach provides the minimum classification/recognition latency, executing the network every time a new sample is available to update the input sliding window, it can determine performance to be bandwidth limited. This is because all network parameters/weights must be loaded for every layer. So, despite the double-buffered scheduling strategy, transfer and computation phases hardly overlap, affecting the *operational intensity* of the application. The roofline

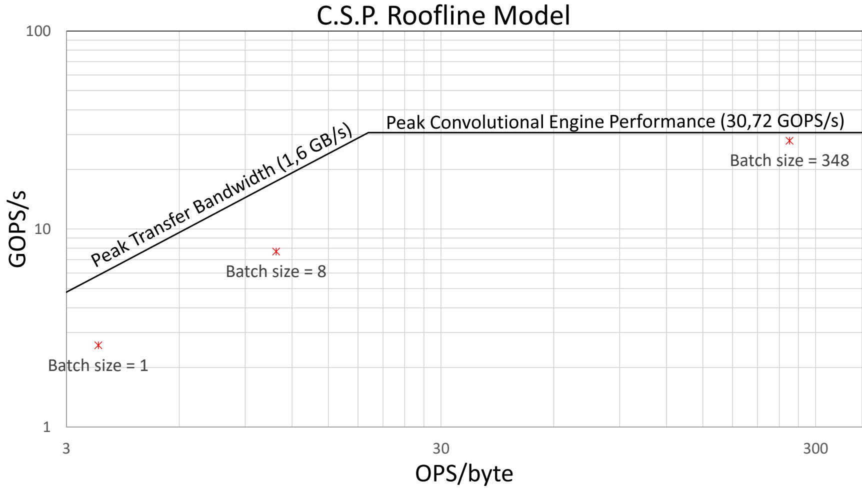


Figure 3.8: Use case benchmark's Roofline Model for the Convolution Specific Processor (12x4 MAC Matrix in Z-7020 SoC) with respect to different batch sizes

model in Figure 3.8 shows performance trends starting from the sample-by-sample processing (leftmost red cross), on a use-case that will be presented in the following section.

If the latency constraint is not extremely tight, it is possible to pre-buffer input samples to process longer sample sequences (sample batching) and produce more output with every execution. To implement such a strategy, sample acquisition and sequence processing were decoupled using a buffer mechanism. For example, an independent task, executed timely, e.g. triggered by a timer, can acquire samples from a sensor and make them available inside an adequate buffer

for further processing. Real-time execution, in this case, requires the processing task to sustain a sufficient throughput to avoid a buffer overrun. Such a sample batching mechanism takes profit from the fact that in sequence processing, different execution of the network operates on successive snapshots of the same sliding window, sharing part of the receptive field. Thus, every iteration reuses part of the intermediate results produced in previous instants to derive the output of the network. This can be exploited with adequate scheduling of the I/O data transfers to improve efficiency. Figure 3.9 shows the transfer scheduling when *batch size* is increased. As the *batch size* increases the architecture gets closer and closer to the computational limit (rightmost red mark on the roofline plot of Figure 3.8) because of the growing number of operations performed. This way, it is possible to increase the utilization of computing resources and to gain efficiency. As it may be noticed in Figure 3.8, when sample batch size is small, besides being on the bandwidth-limited region of the plot, points in the roofline model are also distant from the theoretically achievable performance. This is due to overheads related to CE programming/warm-up, and to initial and final input/output transfers, which cannot overlap with convolutions. The impact of this overhead is limited when the operational intensity increases, reducing the distance between points and the theoretical roofline model.

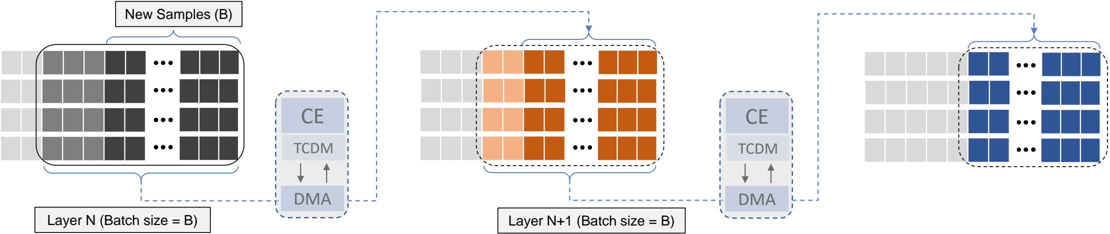


Figure 3.9: TCN execution on NEURAghe: *batch size* = B

3.3 Hardware Implementation Evaluation

3.3.1 Design Space Exploration

A designer willing to use the NEURAghe template, on a given target SoC, has multiple possible architectural configurations available. In order to perform a careful selection, it is possible to perform a simple design space exploration and to choose a near-optimal setup from the performance point of view. To select the architectures presented in this chapter, a simple grid search was used. It evaluates multiple configurations, featuring different values of N_{rows} and N_{cols}, to maximize the number of SoPs, while keeping the number of used DSPs and BRAMs in the range of availability imposed by the target SoC. Table 3.1 shows utilization numbers estimated using the Equations 3.3 and 3.4. Light-grey coloured cells in the table indicate which configurations are not implementable in a Xilinx XC7Z020 SoC, due to excessive DSP or BRAM utilization. Darker-coloured cells indicate configurations that are not feasible in a Xilinx XCZU3EG device. After the design space exploration, it is possible to select, for the considered target device, the configurations that maximize the number of implementable MAC modules, and, consequently, the peak performance of the architecture to be implemented, as the red and blue full line circled configuration in Table 3.1. However, as will be shown in the following, similar configurations with slightly lower utilization of the device could sometimes be clocked at higher frequencies, compensating for a lower count of MAC modules, as, for example, the dashed blue circled configuration in Table 3.1. An empirical evaluation of top design points in terms of maximum sustainable frequency can be used to identify prospectively efficient alternatives among top-utilization configurations.

3.3.2 Implementation on different SoCs

Architecture configurations have been implemented with different Convolution Engine's MAC Matrix shapes, selected after the DSE process, on two target platforms: a Xilinx XC7Z020 and a Xilinx XCZU3EG featuring the Ultrascale+ technology. For the XC7Z020 two configurations have been selected, featuring a similar number of DSP slices and similar clock frequencies. Table 3.2 shows resource occupation on the reconfigurable logic of a first configuration implemented on the XC7Z020, featuring a 12×4 MAC matrix of SoP units, that can be clocked up to 120 MHz, providing peak performance of 46 GOPS/s. Table 3.3 shows results related to a similar configuration that integrates an 11×5 MAC matrix, using slightly more DSPs but clockable up to 110 MHz, with a peak performance of 48.4 GOPS/s. Finally, Table 3.4 describes the results of the implementation of a configuration, featuring a 9×10 MAC matrix of SoP modules, on the XCZU3EG. This design

Table 3.1: RAMB18 and DSP utilization in NEURAghe architecture with respect to the MAC Matrix shape

Each cell: RAMB / DSP

N_{rows} \ N_{cols}	4	5	6	7	8	9	10	11	12
4	144 / 64	156 / 80	168 / 96	180 / 112	192 / 128	204 / 144	216 / 160	228 / 176	240 / 192
5	164 / 80	177 / 100	190 / 120	203 / 140	216 / 160	229 / 180	242 / 200	255 / 220	268 / 240
6	184 / 96	198 / 120	212 / 144	226 / 168	240 / 192	254 / 216	268 / 240	282 / 264	296 / 288
7	204 / 112	219 / 140	234 / 168	249 / 196	264 / 224	279 / 252	294 / 280	309 / 308	324 / 336
8	224 / 128	240 / 160	256 / 192	272 / 224	288 / 256	304 / 288	320 / 320	336 / 352	352 / 384
9	244 / 144	261 / 180	278 / 216	295 / 252	312 / 288	329 / 324	346 / 360	363 / 396	380 / 432
10	264 / 160	282 / 200	300 / 240	318 / 280	336 / 320	354 / 360	372 / 400	390 / 440	408 / 480
11	284 / 176	303 / 220	322 / 264	341 / 308	360 / 352	379 / 396	398 / 440	417 / 484	436 / 528
12	304 / 192	324 / 240	344 / 288	364 / 336	384 / 384	404 / 432	424 / 480	444 / 528	464 / 576

	avail. RAMB18	avail. DSP	selected	out of resources
XC7Z020	280	220		
XCZU3EG	432	360		

uses all the DSP slices on the chip and can be clocked at 180 MHz, providing a peak of 129.6 GOPS/s.

Table 3.2: Resource occupation on a Xilinx XC7Z020 (12x4 MAC matrix)

	DSP	BRAM	LUTs (logic)	LUTs SR	Regs
Used	192	120	47230	259	26942
Available	220	140	53200	53200	106400
%	87.27	85.71	88.78	1.49	25.32

Table 3.3: Resource occupation on a Xilinx XC7Z020 (11x5 MAC matrix)

	DSP	BRAM	LUTs (logic)	LUTs SR	Regs
Used	220	128	42964	283	26962
Available	220	140	53200	53200	106400
%	100	91.4	80.76	1.63	25.34

Table 3.4: Resource occupation on a Xilinx XCZU3EG (9x10 MAC matrix)

	DSP	BRAM	LUTs (logic)	LUTs SR	Regs
Used	360	354	47857	159	26463
Available	360	432	70560	70560	141120
%	100	81.94	67.82	0.4	18.75

3.4 Experimental Results

The actual level of performance reachable using the accelerator has been tested on three different benchmarks, trying to cover the landscape of TCNs presented in literature as much as possible:

- A *plain* TCN network for ECG monitoring and classification (Goodfellow et al. [35]), that performs classification over single lead ECG waveforms and reaches around 90% average accuracy, *ECG* in the following. This benchmark exposes real-time constraints and can be used to assess the usability of our accelerator in this kind of context.
- A TCN for 3D human action recognition (Kim et al [62]), called hereafter Res-TCN, based on residual units, with a structure taken from the Resnet family [44]. In this TCN authors started from a 3D human action recognition dataset with 3D full skeleton annotations to extract a 1D feature representation per frame resulting in a 150-dimensional vector. Applications for this kind of task range from video surveillance, robotics and skill evaluation.
- A more complex network, based on WaveNet (Van Den Oord et al. [99]), for Polyphonic Note Transcription of Time-Domain Audio Signal [71] (WN-PNT in the following).

Each of these tasks, depending on the application context, may require a near sensor data processing. As it happens, for example, in ECG monitoring devices or in video surveillance that pose a privacy issue for which network data sharing is discouraged, or in robotics, where latency for cloud data offload could not be tolerable.

Three system implementations using NEURAghe have been considered, two of them, 11×5 and 12×4, implemented on a Zedboard development board featuring the XC7Z020 SoC, and one implemented on the Ultra96 development board featuring the XCZU3EG MPSoC, with the MAC matrix respectively clocked at 70 MHz, 80 MHz and at 180 MHz, selected to match the speed-grade of the device available on the development boards. For each implementation of each benchmark, performance efficiency has been reported computed as the ratio between measured performance and the peak performance theoretically achievable by the specific configuration (depending on the number of MACs and clock frequency). Single-layer efficiency and average efficiency for the whole network will be shown.

Table 3.5 shows the comparison between on-chip memory capabilities for each system implementation and the memory footprint of each use-case network that involves the overall occupation of weight kernels and input features through all layers, considering the sample batch size values that will be used in the following

subsections. As it may be noticed, all the networks must be adequately managed with the scheduling strategy presented in section 3.2.3, since their activation and weight memory footprint exceeds the memory available on the considered low-cost hardware platforms.

Table 3.5: Implementation related on-chip memory capabilities and use-case networks memory footprint

	input features [kB]			weight kernels [kB]
12x4 on XC7Z020 on-chip memory	192			96
11x5 on XC7Z020 on-chip memory	176			110
9x10 on XCZU3EG on-chip memory	144			180
ECG:	**B=1**	**B=8**	**B=348**	
memory footprint	216,8	250,8	1696,5	11495,6
Res-TCN:	**B=1**	**B=144**		
memory footprint	37,3	511,5		5424,8
WN-PTN:	**B=1**	**B=504**		
memory footprint	553,5	5846,49		3328,8

3.4.1 ECG classification use-case

The network consists of several computational blocks made of a 1D Convolutional layer, a batch normalization layer, a ReLU, a pooling layer and a dropout stage. Layers are followed by a Global Average Pooling and a Softmax function. The network operates on streams of 16-bit samples, acquired at 300 Hz frequency.

Table 3.6: Convolutional Layer characteristics for Network ECG [35]

	type 1	type 2	type 3	type 4	type 5	type 6	type 7	type 8
$input features$	1	320	256	256	128	128	128	64
$output features$	320	256	256	128	128	128	64	64
$Kernel_size$	24	16	16	8	8	8	8	8
$dilation\ rate$	1	2	4	4	6	8	8	8

We have assessed three different operating modes:

- *minimum latency mode*,
- *maximum throughput mode*,
- *real-time execution*.

The first one provides classification in output after minimum latency. The network is executed as soon as possible per every input sample. In the second operating mode, sample batching is used extensively, to maximize throughput without considering latency as an optimization objective. In the third mode, sample batching is exploited just enough to reach a throughput that allows respecting the real-time constraint posed by the input sampling frequency (300 Hz).

Figure 3.10 shows the performance achieved on this benchmark by configurations implemented on the Zedboard and the Ultra96-V2, while Figure 3.11 shows execution times. As expected, without sample batching, performance is significantly bandwidth-limited. In minimum latency mode ($B = 1$), efficiency is very low for every layer, and consequently on the whole network. DSPs have very long idle times and actual performance is very far from the peak. Execution time is around 17 ms with the 12×4 matrix, a bit higher with the 11×5 (20,6 ms), while the fastest version is the 9×10 (7,58 ms) due to the higher clock frequency and the higher number of SoP modules.

To design the maximum throughput mode, an iterative test exploiting different batch size values was performed, in order to identify the values saturating the benefits achievable with sample batching. For this benchmark, such value is B=348, corresponding to a latency of around $1.2s$. In this case, all configurations can operate very near to the peak performance, with an average efficiency of around 0.9 and a best per-layer efficiency of 0.96 achieved by the 12×4 configuration. Some

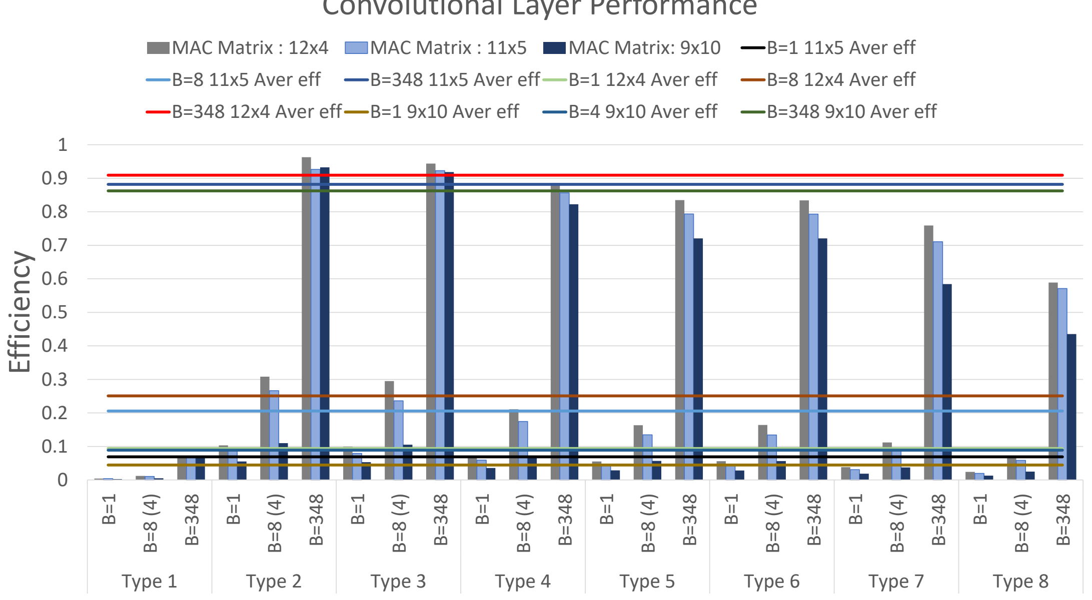
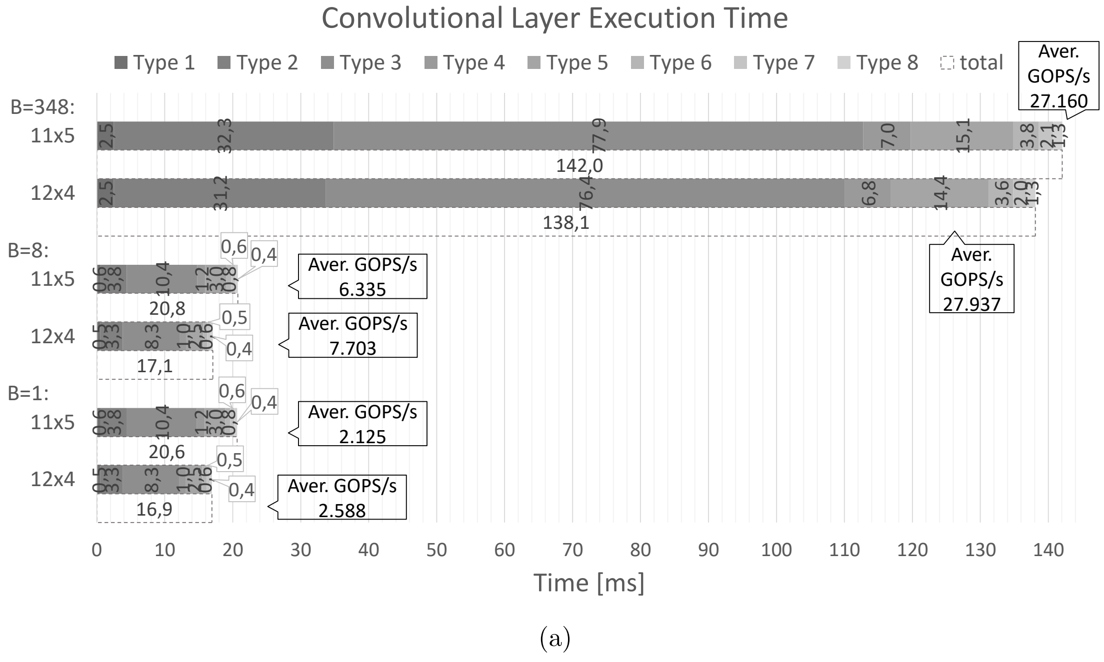


Figure 3.10: Efficiency trend on ECG [35] for different batch sizes for NEURAghe 12×4 and 11×5 MAC matrix configurations in XC7Z020 and 9×10 MAC Matrix configuration in XCZU3EG

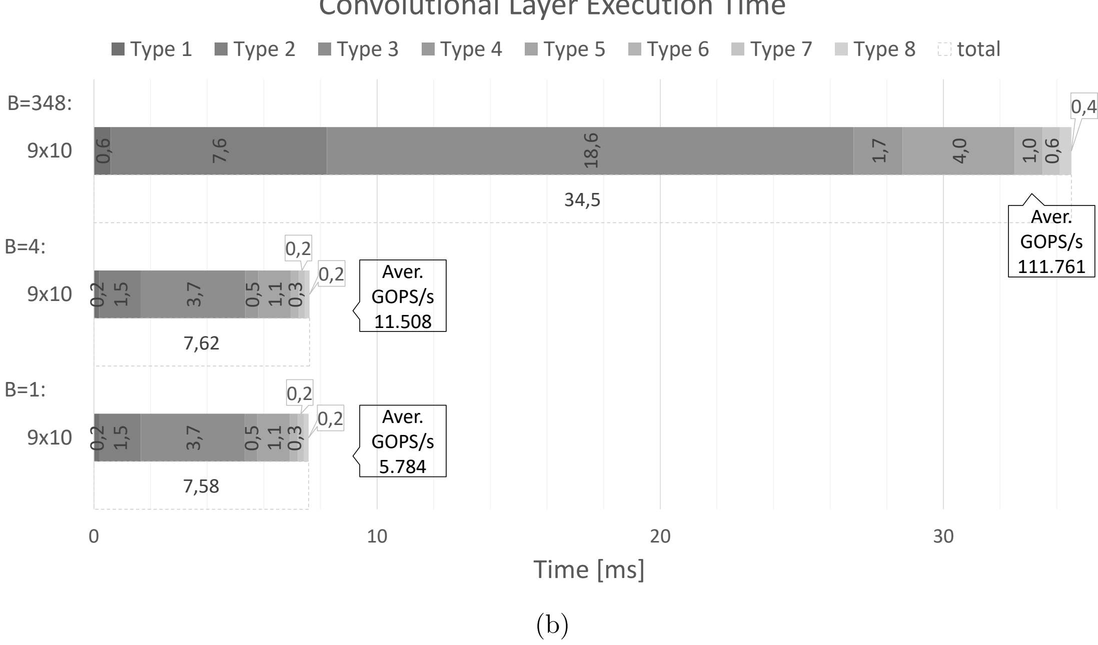
layers, especially Type 1, are still less efficient, however, their contribution to the overall execution time is limited. In maximum throughput mode, for the XC7Z020 implementations, each network execution takes around 140 ms, producing in output classification of 348 sample consecutive sample sequences, corresponding to one sample every $0.4ms$, while for the XCZU3EG execution time is 34.5 ms corresponding to around 0.1 ms per sample.

Finally, the real-time constraint requires processing one sequence in less than 3.3 ms. To design the real-time operating mode, by exploring the batch size, $B = 8$ is identified to be the lowest value enabling compliance on such requirements in XC7Z020 cases. Execution time is around 17 ms, corresponding to around 2 ms per sample. The XCZU3EG implementation can use $B = 4$.

Since 12×4 is more efficient than 11×5 in all modes, in this case, higher frequency is more important than the number of MACs, due to better utilization with the specific layer characteristics (the matrix is partially unused in the final operations of a convolution when the number of output features is not a multiple of 5).



(a)



(b)

Figure 3.11: Execution Time comparison on ECG [35] for NEURAghe (a): 12×4 and 11×5 MAC matrix configurations in XC7Z020 and (b): 9×10 MAC Matrix configuration in XCZU3EG

3.4.2 Res-TCN use-case

This use case considers the execution of the network presented in [62], made of stacked units, composed of 1D convolutions, batch normalization and ReLU layers featuring a residual connection, ending with a Global Average Pooling layer and a Softmax function. Table 3.7 shows the characteristics of the different types of Convolutional Layers in the network.

Table 3.7: Convolutional Layer characteristics for Res-TCN [62]

	type 1	type 2	type 3	type 4	type 5	type 6
$input features$	150	64	64	128	128	256
$output features$	64	64	128	128	256	256
$Kernel_size$	8	8	8	8	8	8
$stride$	1	1	2	1	2	1

Figure 3.12 shows efficiency and execution time on this benchmark, highlighting contributions of the single layers. Its behaviour is similar to the ECG use case on

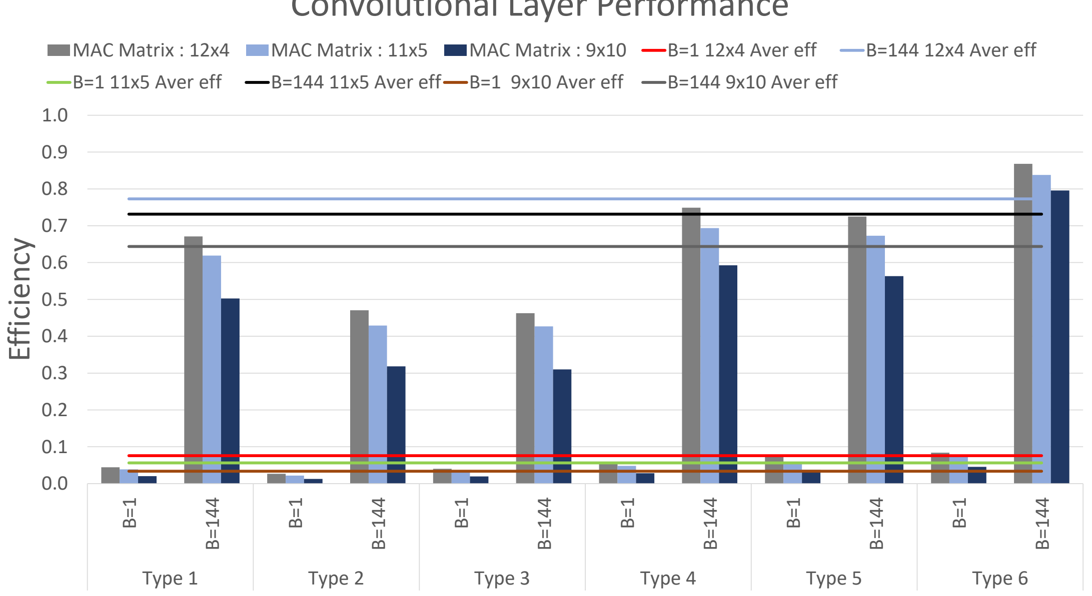


Figure 3.12: Efficiency trend on Res-TCN [62] for different batch sizes for NEURAghe 12×4 and 11×5 MAC matrix configurations in XC7Z020 and 9×10 MAC Matrix configuration in XCZU3EG

both boards. Executing the processing after each sample ($B = 1$), the efficiency is very limited for all the layers. Increasing B, the accelerator provides much better

throughput. The layers executed less efficiently are the early layers, especially the first type, that pays for a significant under-utilization of the MAC Matrix, due to the low number of input features. However, its contribution to the overall efficiency is marginal. Layers that show longer execution time are placed at the end of the TCN graph. These layers account for the highest contribution to the overall workload and, as shown in Figure 3.12, are executed quite efficiently (0.86 as the best per-layer value in 12×4 configuration) when sample batching is used ($B = 144$). Sample batching is more effective on the Zedboard, which integrates a MAC Matrix with less input and output ports, thus requiring longer runs to complete convolution, which reduces the impact of data transfer and accelerator warm-up overheads.

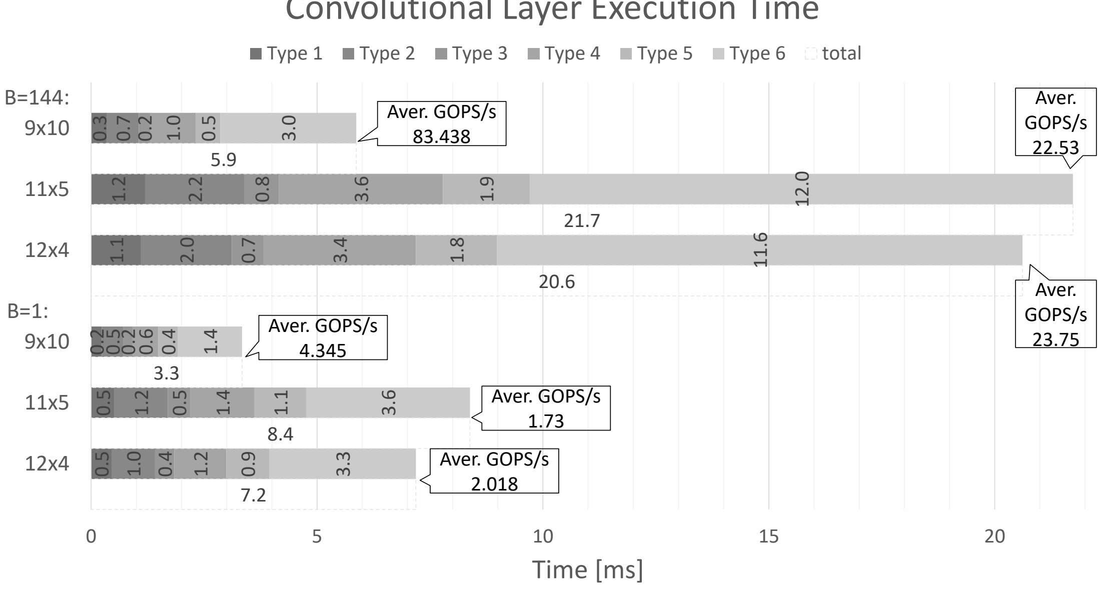


Figure 3.13: Execution Time comparison on Res-TCN [62] for different batch sizes for NEURAghe 12×4 and 11×5 MAC matrix configurations in XC7Z020 and 9×10 MAC Matrix configuration in XCZU3EG

3.4.3 WN-PNT use-case

This case considers a network derived from WaveNet (Van Den Oord et al. [99]) for Polyphonic Note Transcription of Time-Domain Audio Signal [71]. It is made of 20 stacked blocks composed by a 1×1 skip connection and a Dilated Convolutional block of 128 channels each, interspersed with a *sigmoid* and a *tanh* funcions and featuring a residual connection between subsequent blocks. *Filter Size* is 2 and the *dilation factor* grows at 2^k where the residual block index $k \in \{0, 1, 2...9, 0, 1, 2...9\}$.

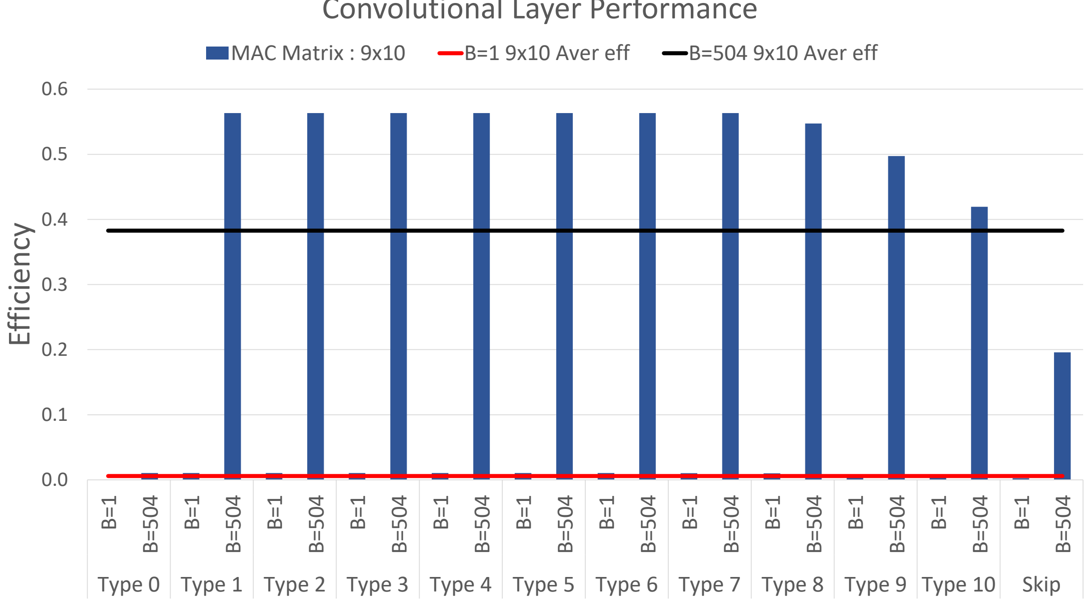


Figure 3.14: Efficiency trend on WN-PNT [71] for different batch sizes for NEURAghe 9x10 matrix configuration in XCZU3EG

This case is especially challenging, due to the small size of kernels ($kernel_size = 2$ in most layers) and because the sample acquisition frequency is $16KHz$, posing tight constraints on the real-time execution. Thus the focus is given on the Ultra-96 board, relying on its higher clock frequency to achieve the required throughput. As may be noticed in Figure 3.14, in general, this use case is executed less efficiently on the platform, none of the layers reaches more than 0.56 efficiency. This is because the execution of the actual convolution kernels takes only two cycles and thus hardly overlaps with input data transfers. Batch size can be used to increase the reuse of weights, once they are loaded to the weight memory region, but at the same time, has an impact on the duration of input and output transfers. To compensate for this issue, considering that the total size of

the local receptive fields in the network allows for the continuous storage on the on-chip memory for almost all layers (except for types 8, 9 and 10), the scheduling described in Figure 1.8 has been slightly modified. In this use case, transfers to/from DDR involve only new output/input samples, while keeping the rest of the local receptive fields in the on-chip memory. As shown in Figure 3.15, using $B = 504$ we can process 504 samples in around $19ms$, thus respecting the real-time constraint required by the use-case.

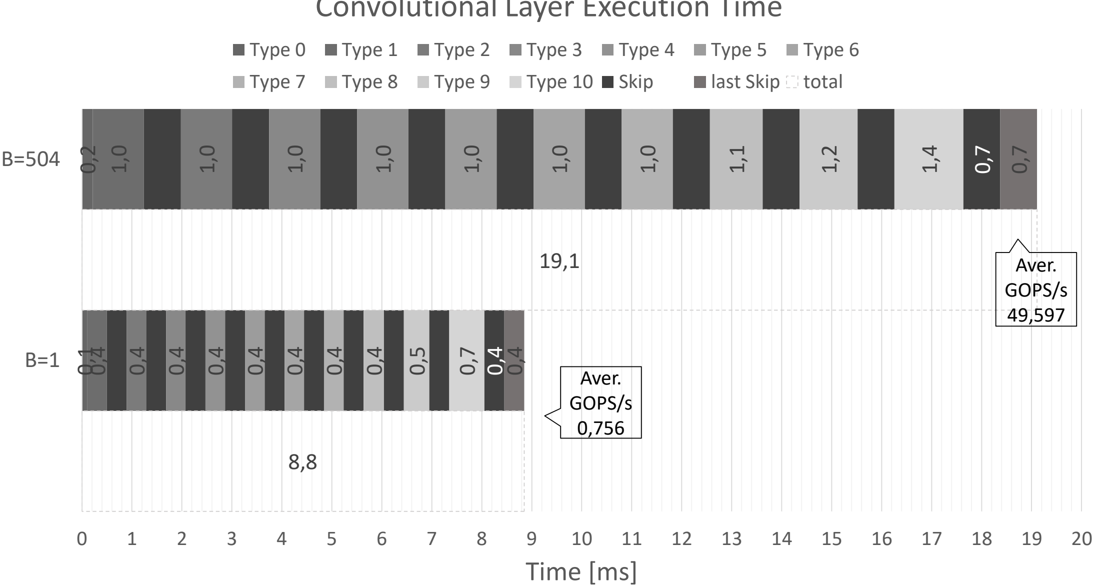


Figure 3.15: Execution Time on WN-PNT [71] for different batch sizes for NEURAghe 9x10 matrix configuration in XCZU3EG

3.4.4 Assessment of hardware vs. software speed-up

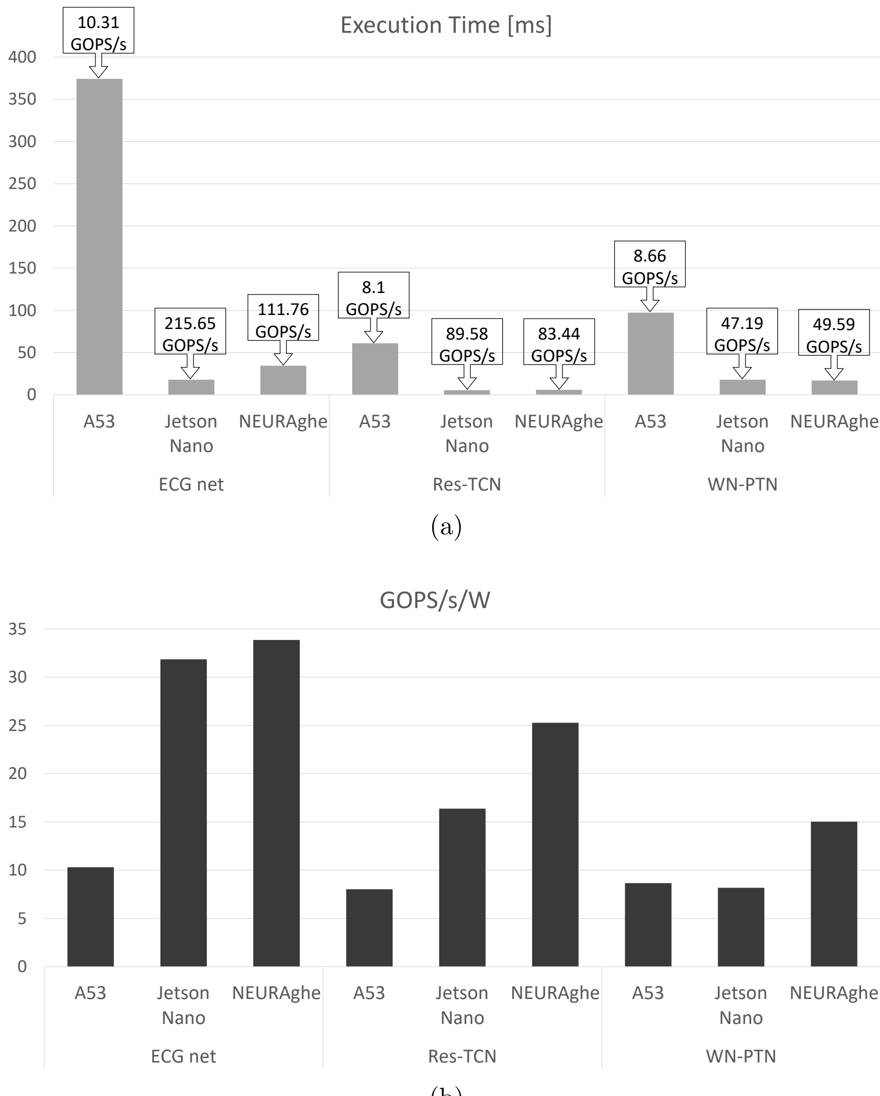
To evaluate the benefits obtained by FPGA acceleration, the execution of the convolution-related load of the three benchmarks on NEURAghe has been compared with two other software-programmable processing systems. The first selected term of comparison is the execution on an A53 quad-core ARM processor, using the ARM Compute Library optimized functions to implement convolution layers[1]. As it may be noticed, NEURAghe provides up to 10.8x, 10.4x and 5.7x speed-up on the three reference benchmarks with respect to the A53, and considering the power consumption of the two platforms (around 0.9 W for the A53 cores and around 3.3 W for NEURAghe[2]), PL-based acceleration provides an improvement, in terms of power efficiency, corresponding to 33,8 GOPS/s/W , 25,3 GOPS/s/W and 15 GOPS/s/W (around 3.5x, 3x and 2x with respect to A53) respectively. The term for comparison has been the Jetson Nano, a GPU-endowed System-on-Module, featuring an NVIDIA Maxwell GPU and a Quad-core ARM Cortex-A57[3]. Figure 3.16a shows that Jetson Nano outperforms NEURAghe in terms of absolute performance (around 2x speed-up) on the ECG net benchmark, which exposes higher operational intensity, mainly due to its higher clock frequency. On the other hand, execution times on the remaining two benchmarks are comparable. Considering the module power consumption, which ranges from around 4W to around 7W depending on the Jetson Nano utilization in a specific layer, NEURAghe still provides higher power efficiency, up to around 2x for WN-PTN. While this experiment confirms that a significant gap exists in terms of overall performance, between the two compared platforms, it also confirms that in some specific benchmarks, improvement of hardware flexibility and data loading strategies may compensate

[1] Since ARM-CL does not support dilated 1D convolution, for the sake of comparison, the evaluated execution time on the A53 on analogous convolution layers has been done without dilation. Considering that in dilated convolutions input samples are not adjacent in memory, the exploitation of the vector processing in the ARM-CL could be sub-optimal, thus the execution time reported in Figure 3.16a for the A53 could be underestimated. Our software implementation (unoptimized) of a dilated convolution performs one order of magnitude slower than what is reported for ARM-CL.

[2] Power consumption for the APSoC has been evaluated using Xilinx Power Evaluator, fed with resource utilization numbers coming from Vivado after the implementation and switching activity figures obtained by RTL-simulation on Mentor QuestaSim. NEURAghe configurations implemented on the Xilinx XC7Z020, with a 12x4 and 11x5 MAC Matrix, have a power consumption of 2.85 W and 2.91 W respectively.

[3] The execution time has been measured executing Conv layers in the reference benchmarks as ONNX [11] operators, using the trtexec utility provided in https://github.com/NVIDIA/TensorRT. Power has been measured accessing the INA3221 power monitor available on the module and by considering tensors in input to layers to be sized according to the proposed sample batching method, to compare over the same number of operations. Time and power consumption values result from an average of 40 successive executions.

for this gap. Moreover, opening TCNs to FPGA-based acceleration will prospectively enable some of the advantages that reconfigurable devices usually provide with respect to GPUs, such as exploitation of custom data formats and aggressive quantization.



(a)

(b)

Figure 3.16: Execution time and power efficiency comparison between software execution on a Cortex-A53 quad-core and NEURAghe (XCZU3EG).

3.4.5 Comparison to other APSoC based accelerators

As mentioned, the literature is lacking TCN evaluations on FPGA-based accelerators. The only attempt available at the time of the work depicted in this chapter was that of Hussain et al. [49], which is strictly customized for an autoregressive TCN and is implemented on a high-end board not usable in the embedded domain. However, the design of the TCN-supporting version of NEURAghe considers the compatibility with 2D convolutions for classic CNN acceleration. Moreover, the improved flexibility of the architecture, in terms of kernel size and stride, allows for some improvements with respect to other approaches in the literature targeting low-to-mid-end all-programmable SoCs. Table 3.8 reports comparative results with the state-of-the-art on two well-known networks for image classification, ResNet-18 and VGG-16, and to other accelerator architectures, (including the version presented in Chapter 2) [89] and [102], that are implemented on the same kind of hardware and use the same 16-bit data precision[4]. On VGG-16, which exposes quite regular kernel sizes and stride values, this work shows comparable performance with respect to the alternatives. It executes convolutions slightly faster than the accelerator version presented in Chapter 2 and the work in [89] and around 13% slower than [102]. On ResNet-18, which exposes more variable *kernel sizes* and *strides*, a lot of overheads that must be paid by more *static* architectures can be reduced, executing the whole convolution workload 40% faster than the version in Chapter 2.

Table 3.8: Comparison between NEURAghe version describend in this chapter (NEURAghe TCN), the previous version (Chapter 2) and other works on ResNet-18 and VGG-16. Xilinx XC7Z020

	NEURAghe TCN	NEURAghe Chap.2	**NEURAghe TCN**	NEURAghe Chap.2	[102]	[89]
	ResNet-18		VGG-16			
Xilinx Zynq SoC	**XC7Z020**	XC7Z020	**XC7Z020**	XC7Z020	XC7Z020	XC7Z020
Freq. [MHz]	**120**	120	**120**	120	125	150
GOPS/s	**26,5**	16,1	**42,62**	42.48	48.53	31.38

[4]Comparison concerns implementation on XC7Z020 only as it was one of the target SoC for the referenced works and also the NEURAghe version depicted in Chapter 2

3.5 Summary

This chapter presents the improvements made to the accelerator architecture NEURAghe, in supporting Temporal Convolutional Networks. To serve this specific computing pattern some features have been introduced, such as the capability of supporting arbitrary *kernel_size*, *dilation rate* and *stride values* without overhead. We show also how data transfers from-to off-chip memory can be managed in TCNs and how performance improves by changing the computational paradigm, going from a *latency constrained* approach to a *batched* approach, that trades off latency for throughput. This method has been applied to three notable TCNs and uses two different SoCs as a target, analyzing throughput, latency and efficiency results, and the capabilities of the system to respect real-time constraints. Results show that using sample batching, it is possible to achieve efficiency up to 0.96, 0.86 and 0.57 on the three use-cases, with respect to the peak achievable by each configuration. In the two use-cases with real-time requirements, adequate sample batching can be used to achieve sufficient throughput to timely process all input samples. Comparing the execution of the use-cases on a Cortex A53 quad-core and on an NVIDIA Maxwell GPU, to evaluate the achievable speedup, it is possible to notice, with respect to the first, up to 10x execution time reduction, in NEURAghe, with an improvement in power efficiency that ranges from 2x to 3.5x depending on the benchmark and, for the second, a power efficiency improvement up to around 2x, despite NEURAghe being outperformed in terms of absolute performance. Finally, the compatibility evaluation of proposed architectural solutions with state-of-the-art CNNs used in the image processing domain shows similar performance with respect to CNN-targeting alternatives when regular network topologies are targeted and showing up 40% improvement when targeting more irregular patterns.

4 | Adapting for mobile network operators

As seen in the introductory chapters, the high computing intensity required from classical CNNs justified the development of specific platforms to support their execution. However, the deployment of such networks in tiny embedded devices is often not an easy task. Recent approaches for edge-oriented and mobile applications seek to reduce the number of network parameters, i.e. its memory footprint, fostering the development of different types of convolution schemes. Following this trend, a new group of networks, called LightWeight CNNs (LW-CNNs) ([20, 51, 48, 84, 97]), emerge with the advantages of faster inference time and smaller model sizes compared with conventional CNNs. While LW-CNNs dramatically cut down on the model size, they also introduce new operators, like Depthwise Separable convolutions and Inverted Residual blocks (MBConv) which bring with them an intrinsic inefficiency when processed by conventional CNN accelerators.

In the following subsection, we will explore possible non-invasive solutions to adapt the NEURAghe architectural template towards the Depthwise Separable convolution support. Subsequently, we change the traditional convolutional pattern while designing a new IP to support the Depthwise separable operator and MBConv. Results show how this new pattern allows reaching better efficiency with respect to the theoretical peak performance achievable compared to the non-invasive solutions.

4.1 Depthwise Separable Convolution and MB-Conv

Depthwise separable convolution (DW_Sep) is a form of factorized convolutions that factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution (PW).

While in a standard convolution the two phases of filtering and combining inputs in a new set of outputs are done in one step, as already shown in Section 1.2 (Figure 1.1), in a DW_Sep convolution these phases are separated, resulting in two layers (Figure 4.1).

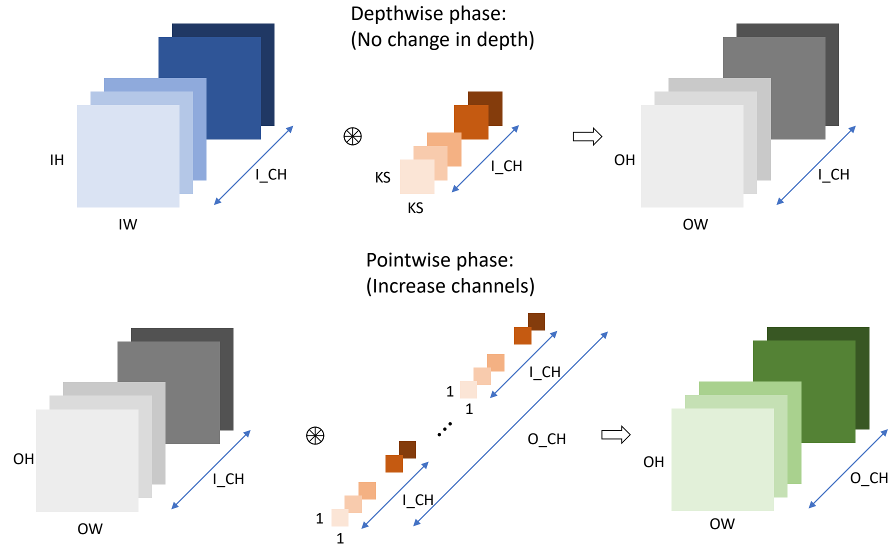


Figure 4.1: Depthwise Separable Convolution

In the first one, a filter is applied to each input channel (I_CH), without any summation, resulting in an output feature with no changes in depth. The second is substantially a standard convolution with output_channel (O_CH) set of filters of size $1 \times 1 \times I_CH$, used to create a linear combination of the output of the depthwise layer, generally increasing the number of channels. The advantage in using this decomposition is in the reduced computational cost and a reduced number of parameters.

A standard convolution layer takes as input a $IH \times IW \times I_CH$ feature map and produces a $OH \times OW \times O_CH$ feature map by applying a $KS \times KS \times I_CH \times O_CH$ convolution kernel, where $IH \times IW$ and $OH \times OW$ are the height and the width of the input and output feature and $KS \times KS$ are the filter size, while I_CH and

O_CH are their depth. In this case, the computational cost is:

$$OH \times OW \times I_CH \times KS \times KS \times O_CH \tag{4.1}$$

In a Depthwise Separable Convolution the depthwise phase has a computational cost of $OH \times OW \times I_CH \times KS \times KS$, while the pointwise phase has a cost of $OH \times OW \times I_CH \times 1 \times 1 \times O_CH$, resulting in:

$$OH \times OW \times I_CH \times (KS \times KS + O_CH) \tag{4.2}$$

which leads to a computational cost reduction of:

$$\frac{OH \times OW \times I_CH \times (KS \times KS + O_CH)}{OH \times OW \times I_CH \times KS \times KS \times O_CH} = \frac{1}{O_CH} + \frac{1}{KS \times KS} \tag{4.3}$$

Moreover, the number of parameters goes from $KS \times KS \times I_CH \times O_CH$ to $I_CH \times (KS \times KS + O_CH)$. Given these characteristics, if properly used, this technique could lead to the above-mentioned advantages with only a small reduction in accuracy ([48]).

MobilNetV2 [84] is an example of such a lightweight model. It relies on particular layers called Inverted Residual blocks or Mobile Bottleneck layers (MBConv) which is a configuration where a DW_Sep layer is preceded by a PW step.

Following this trend, the possibility of including this functionality directly within the architecture shown in previous chapters, with a few modifications has been assessed. As it will be shown, due to its own configuration, handling DW_Sep offloading on the described configuration would not be particularly advantageous (Section 4.2).

With the aim of exploiting the advantages of MBConv layer (and DW_Sep layer if required), while trying to avoid such inefficiencies, we design a new IP that specifically supports these types of layers, resulting in an architectural template for MBConv acceleration with a design-time selectable functionality. Moreover, we introduce a scheduling scheme that maximises the use of Digital Signal Processing Slices, especially when working on the PW phase.

4.2 Exploring unobtrusive solutions

For a Depthwise separable convolution implementation in NEURAghe this operation could be considered in its double phase nature: a depthwise phase and a pointwise phase. The second one is a standard convolution with mono-dimensional kernel size and it is already supported. For the depthwise phase, without introducing too much congestion through new hardware components, NEURAghe management could be performed as shown in Figure 4.2. In particular, for each processing phase, a batch of N_COL input features and weight kernels feeds the MAC matrix and produces N_COL output feature ready to be stored back without further elaboration.

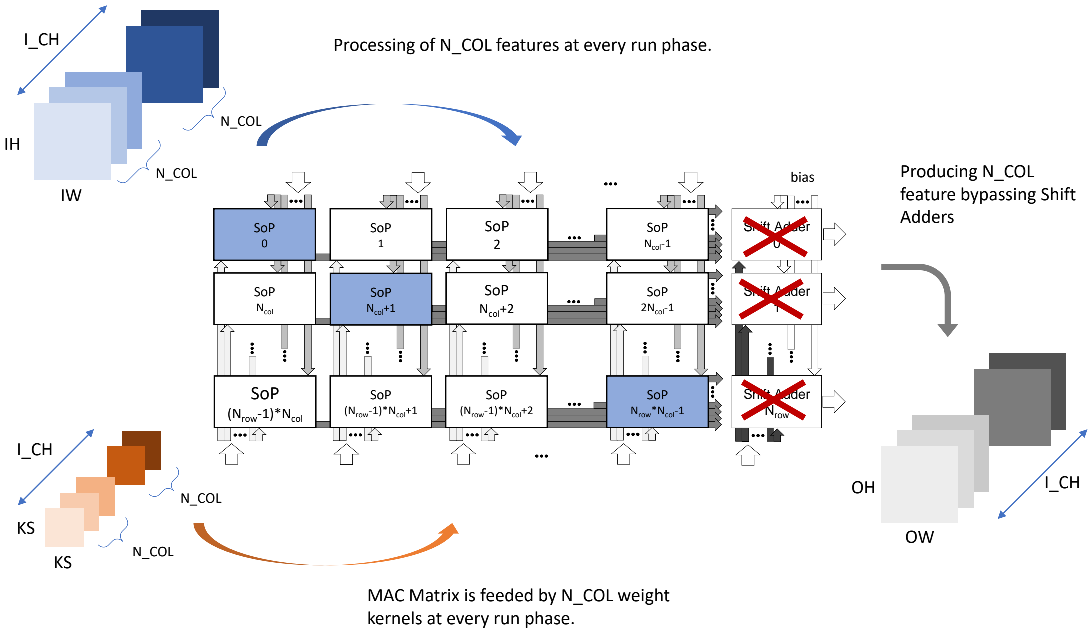


Figure 4.2: NEURAghe DW phase handling

Unfortunately, although this method introduces very few modifications, it leads to a poor MAC Matrix utilization that will result in very poor performance. To face this problem with non-invasive modifications, we identified two possible solutions.

First Solution: SoPs re-configuration

Trying to exploit reuse made possible by the DW_Sep convolution with both phases, NEURAghe and in particular, its MAC Matrix can be slightly reorganized, introducing a few modifications to be activated while processing these types of layers. One possible solution is the one summarized in Figure 4.3. As shown, each Sum of Product unit could be equipped with some more logic to let the matrix manage different execution phases. The bottom of the figure shows a possible operation scheduling.

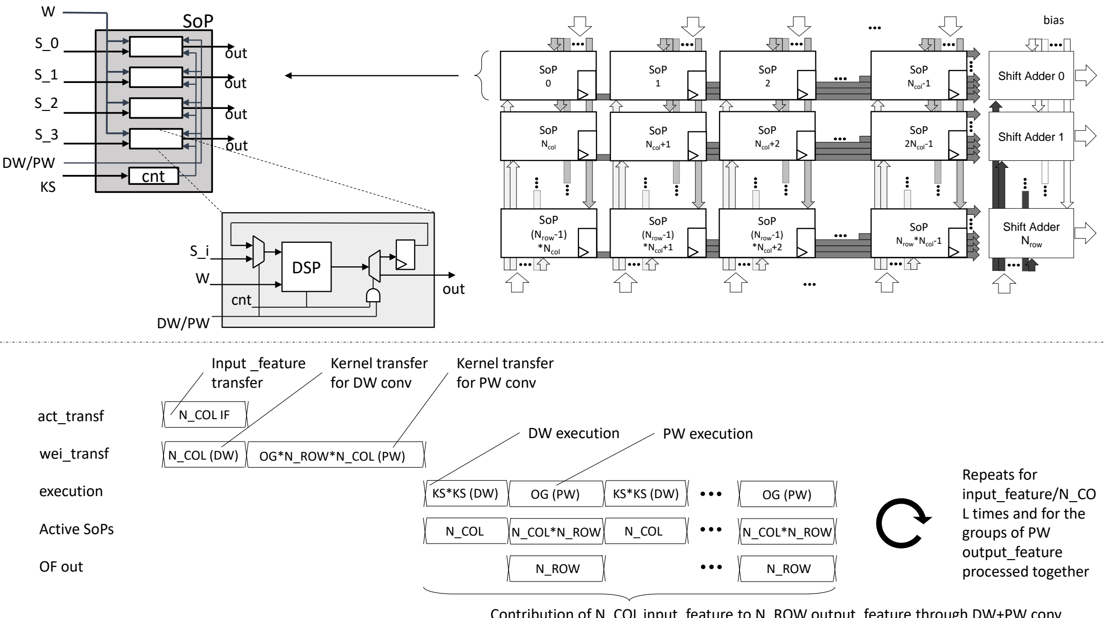


Figure 4.3: Exploiting depthwise separable reuse in NEURAghe: full matrix

At first, N_COL input features and depthwise weight kernels are loaded into activation and weight memory banks ready to feed the matrix. Originally, each SoP that belongs to a column was fed with the same input feature so that it was possible to calculate its contribution to N_ROW output features. In the case of depthwise convolution, each input feature contributes to just one output feature of the depthwise layer. In this case, without introducing further hardware complications, it is possible to spread the same N_COL input feature to each row of a column. Once the data are loaded, the depthwise execution phase can start. For each column (feature), NEURAghe will take $KS \times KS$ cycles to produce 4 samples by applying four depthwise convolution kernels to four adjacent convolution windows. To avoid an immediate store and a future re-load, each new sample, once ready, could be saved in a SoP inner register in charge to re-feed the DSP during the pointwise phase. To this aim, simultaneously with the depthwise execution,

a batch of pointwise kernels can be loaded in weight memory and then used to calculate the contribution of the 4 registered samples to other 4 samples belonging to a number of OG composed by N_ROW output features, in the way the MAC matrix is already able to do.

This process must be repeated for each different N_COL group of input features to obtain their overall contribution to the output groups of N_ROW output features and then iterated for other groups until all output features have been obtained. Despite the high resource utilization, this method implies some drawbacks. For example, during the depthwise phase, although all SoPs are active, their only effective contribution is limited to a row of N_COL, because others are performing the same operation on the same data. This results in $KS \times KS$ inactivity cycles for $(N_ROW - 1) \times N_COL$ SoPs which returns $\frac{OW \times OH}{4}$ times for the number of N_COL groups of input feature and for each of the OG groups composed by N_ROW output features.

Second Solution: SoPs partial re-configuration

Another possibility is shown in Figure 4.4. The idea, when a DW_Sep layer must be processed, is to split the matrix up into two parts and dedicate a row to the depthwise phase while the rest of the matrix is for the pointwise phase. In this case, only N_COL SoPs must be modified to manage depthwise sample buffering and propagation towards the sub-matrix so that it can calculate their contribution to $N_ROW - 1$ output samples.

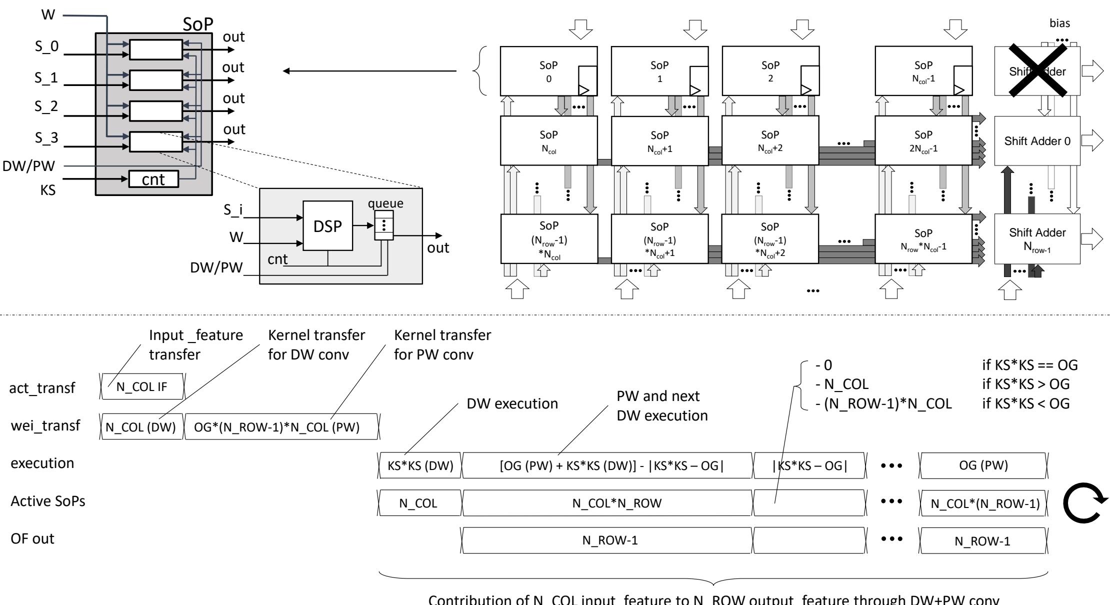


Figure 4.4: Exploiting depthwise separable reuse in NEURAghe: DW sub-matrix

The scheduling of the operations starts the same way as in the previous solution. Once the first depthwise samples have been calculated by the N_COL SoPs, they must be stored in a queue (the reason will be clarified in a while) from which they can feed the sub-matrix for pointwise convolution. After that, both a new depthwise and pointwise processing can start. The first one has a duration of $KS \times KS$ cycles while the second lasts OG cycles. If $OG = KS \times KS$ then the execution iterates with the full matrix active until all the groups of N_COL have been processed to calculate their contribution to OG groups composed by $N_ROW - 1$ output features (last cycles concern only remaining OG pointwise operations).

Otherwise, if $OG \neq KS \times KS$, another group of cycles must be taken into account. The first one is for $KS \times KS > OG$, in which there will be $KS \times KS - OG$ cycles where only N_COL SoPs are active and the sub-matrix must wait. The second is for $KS \times KS < OG$ in which there will be $OG - KS \times KS$ cycles in

which the depthwise N_COL SoPs can start processing the following samples to be stored in the queue. The depth of the queue is given by $\frac{OG}{KS \times KS}$ rounded up to the first integer.

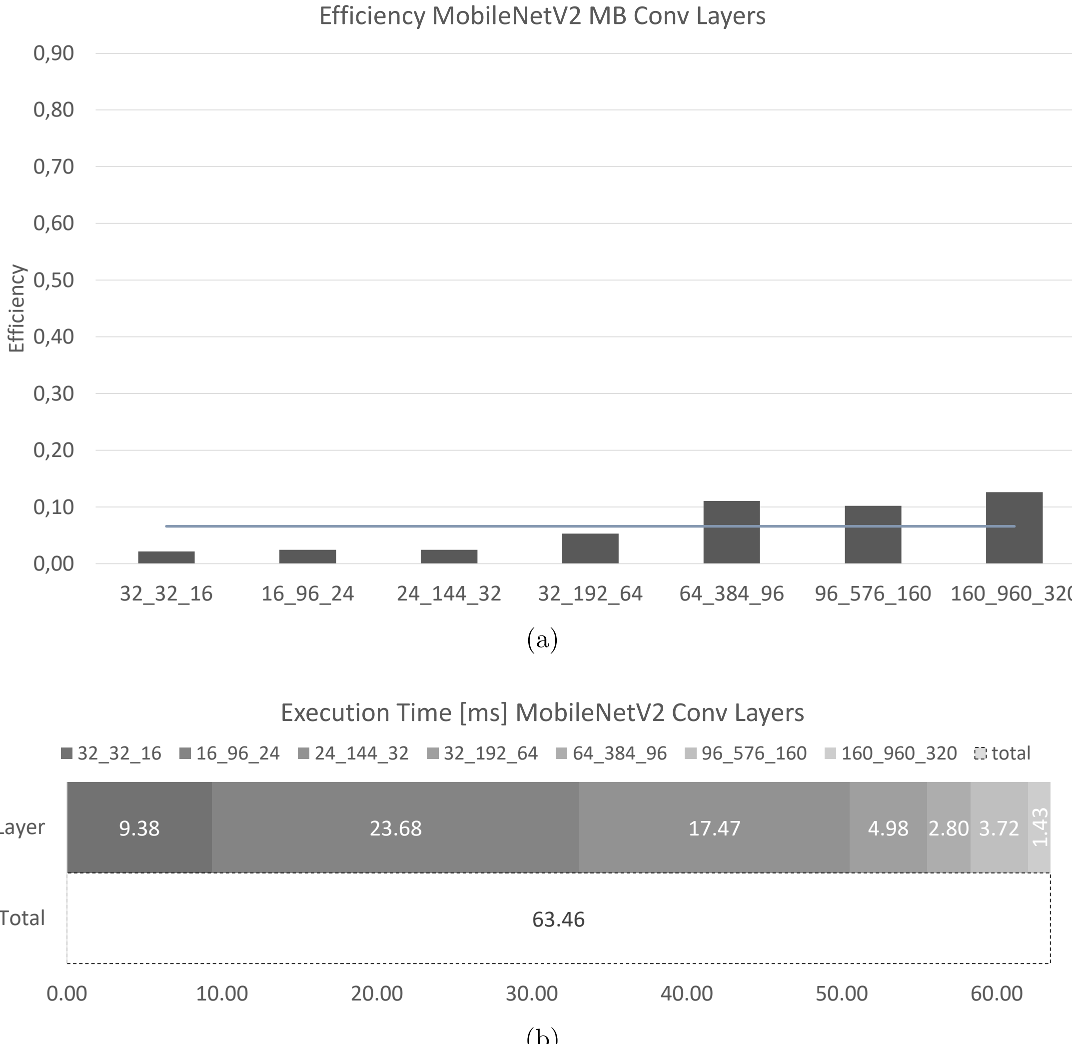
For this last solution, there will be the whole matrix active except for the final queued samples to be processed. In both cases, an MBConv layer execution is fully supported if a PW phase execution precedes the DW_Sep execution as depicted above.

Unfortunately, even if these solutions seem to be a good trade-off between adaptivity (with the pre-existing hardware) and execution capabilities, they hide some drawbacks from the maximum performance with respect to the peak achievable by the architecture. Indeed, apart from MAC Matrix under-utilization in the depthwise phase, there is an inherent inefficiency in the pointwise phase handling if compared to a convolution with an $n \times n$ kernel. Surely for a 1×1 convolution kernel transfers are faster compared to an $n \times n$, but the consumption rate is faster too as the number of operations is smaller: $\frac{1}{2 \times n \times n}$. Besides, considering the DSP utilization, their capabilities are halved as the pointwise phase does not need an inter-kernel accumulation so only the multiplication part of a MAC operation is used. All this leads to a situation where the global per-layer execution time is faster if compared with its equivalent with a standard convolution but it still results in poor efficiency due to the available processing power under-utilization and the shorter amount of time elapsed in a computation phase which is not able to overlap transfer phases and architectural overheads. The situation will not improve considering an MBConv layer, namely an additional pointwise phase with its own transfers between internal and external memory. Table 4.1 shows MobileNet-V2 MB Conv layer characteristics with their multiplicity, feature dimensions, kernel size for the DW phase, input and output channels and stride. Each layer is identified by: *(IN-chan)_(DW-chan)_(OUT-chan)* where DW-chan is the number of channels for the DW phase.

Table 4.1: MobileNet-V2 MBConv layer characteristics

	input	kernel size	output channels	#	stride
32_32_16	$112^2 \times 32$	3	16	1	1
16_96_24	$112^2 \times 16$	3	24	2	2
24_144_32	$56^2 \times 32$	3	32	3	2
32_192_64	$28^2 \times 32$	3	64	4	2
64_384_96	$14^2 \times 32$	3	96	3	1
96_576_160	$14^2 \times 32$	3	160	3	2
160_960_320	$7^2 \times 32$	3	320	1	1

Processing these types of layers with the outlined architectural choices confirms the previous discussions about achievable performance in terms of efficiency, as depicted in Figure 4.5a. Results are obtained by executing on one of the architectural configurations introduced in the previous chapter, with a 9×10 MAC matrix of SoP modules and synthesised on the Xilinx Zynq Ultrascale+ ZU3EG clocked at 180 MHz.



(a)

(b)

Figure 4.5: Efficiency trend and execution time on Mobilenet-V2 MBConv layers

4.3 Specific support for lightweight operators

Trying to avoid the issues highlighted in the previous section while exploiting full DSP capabilities, we have chosen to design a new design-time selectable Convolution Engine IP to natively support DW_Sep and MBConv layers. One of the key strategies has been processing a batch of samples belonging to an input feature map considering its depth rather than processing every single feature according to its horizontal dimensions. In this way, it is possible to exploit both the multiplier and the adder within the DSP units even during the pointwise phase. Furthermore, performing unnecessary transfers between phases for MBConv layers are avoided with a per-phase dedicated MAC Matrix. Figure 4.6 shows the architecture configuration.

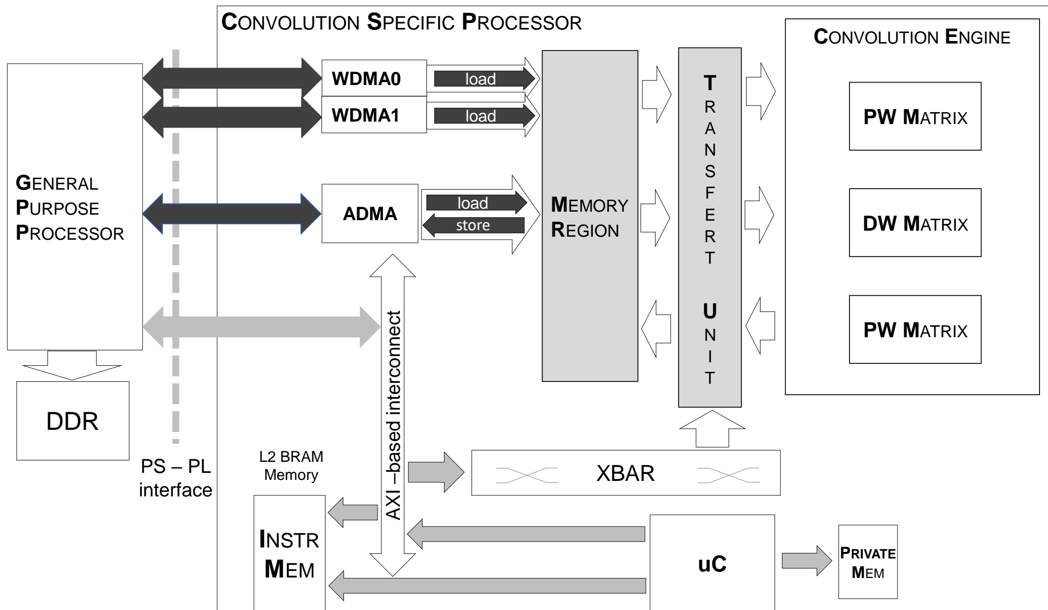


Figure 4.6: Architecture Overview

Thanks to the particular PW processing methodology, as soon as a batch of samples have been processed for all the feature depth, a batch of complete output feature samples is ready to feed the subsequent DW phase that in turn will feed the last PW phase, arranging different phases in a pipeline fashion.

Figure 4.7 shows the Convolution Engine inner organization. It is composed of three Processing Unit Matrices, one for each MBConv phase, fed by filter and input memory banks. A Transfer Unit module orchestrates transfers between memory banks in the memory region.

Each Matrix is composed of $N_{rows} \times N_{cols}$ Processing Units wrapping a DSP

Slice and some extra circuitry to handle saturation on successive accumulations during a MAC operation. It receives samples from N_{cols} input feature maps (IF) to produce samples for N_{rows} output feature maps (OF).

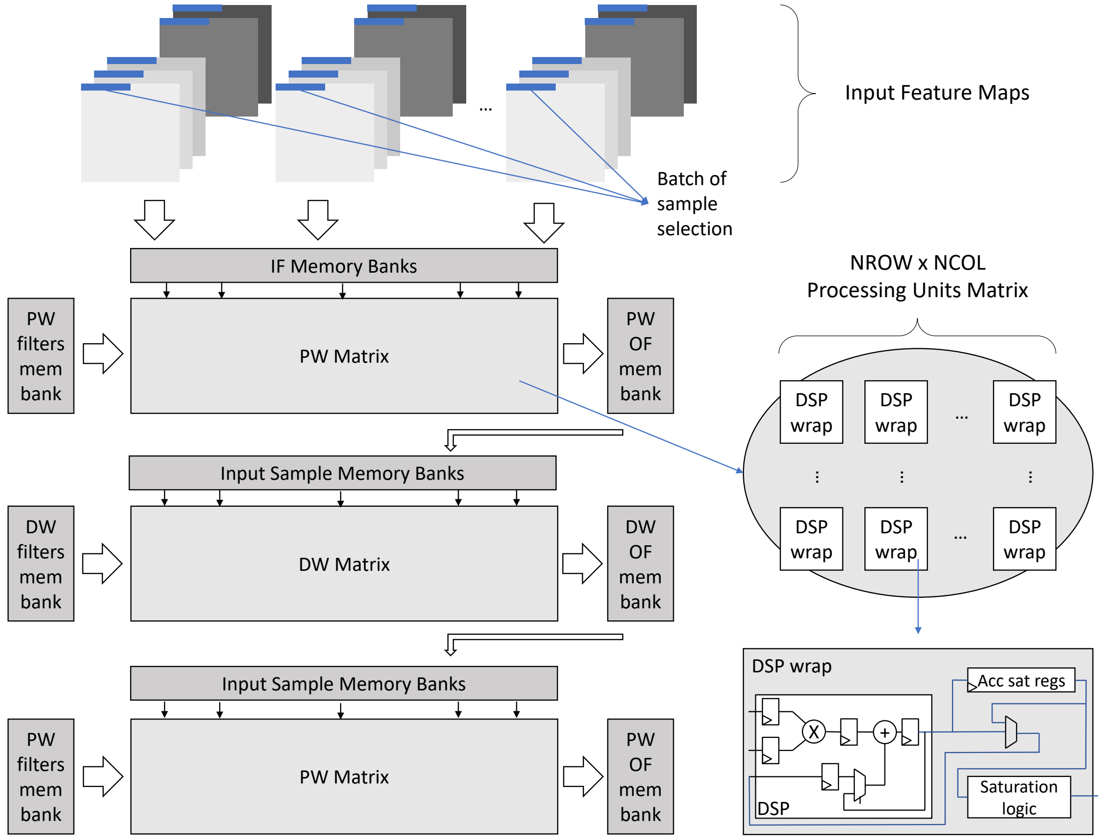
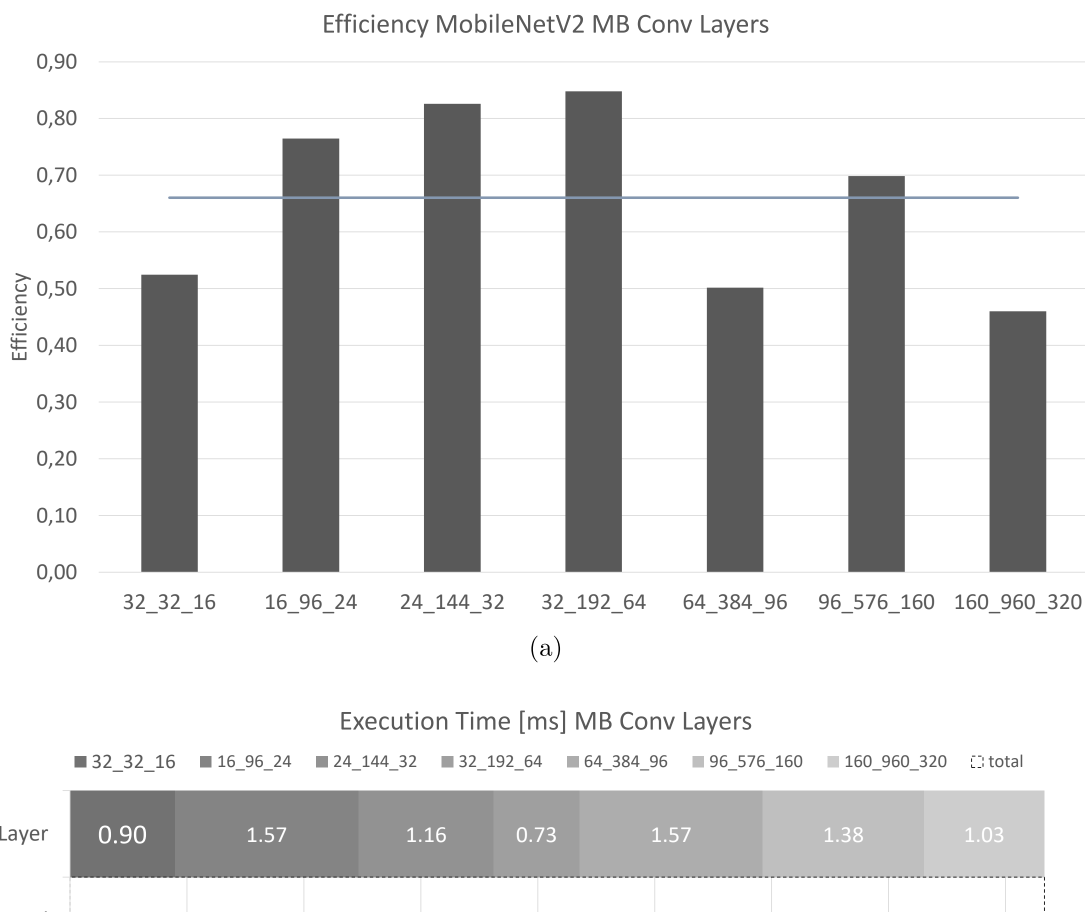


Figure 4.7: Convolution Engine Organization

Figure 4.8 shows the efficiency trend and execution time on Mobilenet-V2 MB-Conv layers, considering an implementation of the described architecture on an Avnet Ultra-96V2 development board featuring a Xilinx XCZU3EG MPSoC. The whole system is clocked at a frequency of 150 MHz thus being able to reach an average efficiency of 67% with respect to the theoretical peak performance of 97,2 GOPS/s considering that the proposed implementation exploits 324 (90%) of the actual available DSP slices in the Programmable Logic of the board.



(a)

(b)

Figure 4.8: Efficiency trend and execution time on Mobilenet-V2 MBConv layers processed with the specific engine

4.4 Summary

Motivated by the growing diffusion of the so-called LW-CNN, particularly attractive in edge-related domains, this chapter presents an assessment of the feasibility of introducing the Depthwise Separable and MBConv support within the NEURAghe architectural template with a non-invasive approach. After an analysis of possible solutions implementing different hardware strategies, we conduct a performance evaluation resulting in a poor efficiency achievable and thus motivating the implementation of a custom accelerator IP that is design-time deployable. This custom IP relies on dedicated MAC Matrix engines to deal with different execution phases and dedicated memory resources to avoid unnecessary data transfers. Moreover, thanks to a smart feature handling it enables a better exploitation of FPGA Digital Signal Processing slices resulting in a notable performance improvement in terms of efficiency and execution time.

5 | Spiking Neural Engine to FPGA adaptation

In the field of edge computing, where a near-sensor efficient computation must be performed, both devices and models must evolve in the direction of lower power consumption with high computational efficiency. In such a scenario, the third generation of Artificial Neural Networks (ANNs) emerged, namely Spiking Neural Networks (SNNs), whose processing paradigm is related to incoming events thus generating an event-proportional workload, required bandwidth and energy consumption. They are biologically inspired neural networks composed of Spike Neurons characterised by an activation behaviour related to their internal state that propagate information by using spikes. Unlike what happens for conventional CNNs, where the activation function of a neuron define its output given the input, in an SNN the output depends on a mathematical model that defines an internal state (membrane potential), a decay constant and a firing rule, namely, a threshold that the membrane potential must surpass to allow an output production. Although more complex mathematical models such as Izhikevich [53] can accurately model a biological neuron's behavior, simpler models such as Integrate and Fire (IF) and Leaky Integrate and Fire (LIF) are more prevalent in current SNN applications ([54],[78],[68],[56]). Given their nature, data in SNNs are represented by binary values, and input and output feature maps are binary tensors, where the presence of a non-zero value represents a neuron spike produced at a certain time.

The accuracy level achieved by this class of networks has become comparable with classical networks [87].

Despite its benefits, SNN characteristics such as reduced computation waste given by event-based processing and the derived data sparsity require a dynamic event managing that creates a run-time dependency and therefore potentially irregular memory accesses, resulting in more difficult parallelism exploitation compared to CNNs.

Because of this, traditional computing architectures are not ideally suited for SNNs. The situation for CPUs, already lacking in performance for the CNN task,

becomes even worse in an event-driven scenario. Also GPUs, that can exploit the network parallelism, are not suited to event-driven computation. This makes the deployment of hardware support for SNN inference particularly challenging making near-sensor neuromorphic computing not simple.

In such a context, the Digital Circuits and Systems group of ETH Zurich proposes a Spiking Neural Engine, SNE, an ASIC targeting highly parallel and modular computational engine designed with the aim to efficiently accelerate SNN inference at the edge. This accelerator is capable to perform parallel execution exploiting an explicit input event address encoding to maximize input data and weight reuse and reduce the temporary data memory footprint.

To enable the benefits allowed by the FPGA flexibility and its fast prototyping capabilities, in the last part of the work presented in this thesis, the collaboration with the above-mentioned group, laid the foundations for the deployment of this accelerator to FPGA. In particular, by applying the NEURAghe model also to the SNN execution results in an FPGA-based SNN inference accelerator capable to process different network topologies without reconfiguration.

In the next sections, we will show some SNE key features as the neuron model and the general architecture organization. Subsequently, we will show key features for accelerator's porting such as standard cells to FPGA's common slices mapping and parameter scaling as a result of overall architectural analysis. Lastly, integration results will be reported in terms of FPGA resource utilization with respect to an "as it is" configuration, maximum operating frequency and performance with respect to the peak achievable by the particular implementation for a use-case experiment.

5.1 Spiking Neural Engine

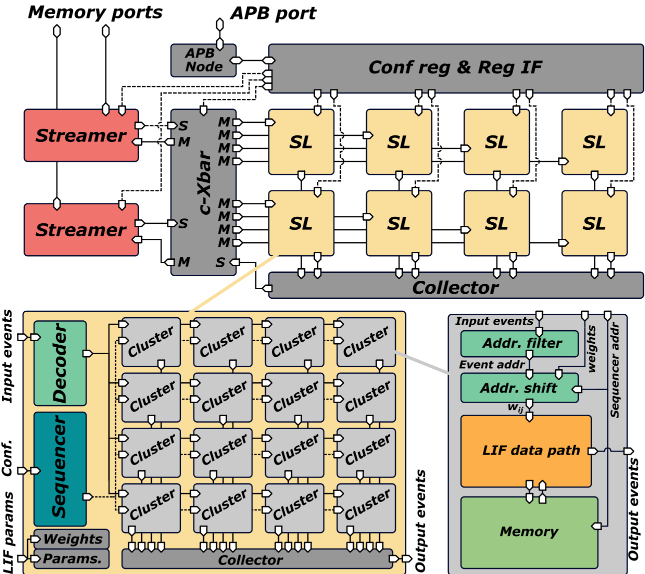


Figure 5.1: SNE architecture overview

Figure 5.1 shows a general overview of the SNE architecture. It is made up of a parametric set of computational engines, called *slices*, that operate in parallel. Slice data transfers are performed by two *streamers* (DMAs) through a combinatorial synaptic crossbar (*c-xbar*). A port is in charge to carry programming for internal registers making the accelerator configuration similar to a memory-mapped peripheral, which exploits the Advanced Peripheral Bus (APB) protocol[1]. Each slice consists of a parametric number of computational units called *clusters* containing the neuron data-path designed to calculate a neuron state update in a single clock cycle. Clusters can operate also as multiple neurons by exploiting a time-domain multiplexing technique that leverages a local buffer to store neuron states. In this version, the local state memory can contain up to 64 neuron states (8-bit wide), resulting in 64 neurons per cluster. Synaptic weights are 4-bit wide. The neuron model implemented is the Leaky-Integrate and Fire (LIF) model with an approximate exponential membrane potential decay. Below is the deployed equation for the membrane potential update (5.1) and the firing rule (5.2). L is the leakage quantity subtracted at every time step, Θ is the Heaviside step function and V_{th} is the threshold.

[1]That is part of the Advanced Microcontroller Bus Architecture (AMBA) family [1]

$$V_{mem}[t+1] = -L + \sum_{j} W_{ij} S_i[t] \tag{5.1}$$

$$S[t] = \Theta(V_{mem}[t] - V_{th}) \tag{5.2}$$

As the SNN with LIF model acts as an Event-based CNN, namely, a standard convolution with the additional time dimension propagated through the layers, the convolution should be performed at each time step updating the neuron state variable. To optimize execution, in SNE, input events are encoded explicitly, so that it can consume single events instead of sliding the entire tensor and output neurons are updated according to explicit addressing by a direct filter selection from a filter buffer. An SNE input event is a 32-bit word composed of two bytes for the x and y feature positions respectively, a byte for the time reference, 4 bits as operation identifier and 4 bits as the channel identifier. As said, slices and cluster are parametric SNE elements, in particular, the configuration under study is exactly that which is shown in Figure 5.1 with 8 slices each composed by 16 clusters.

5.2 Mapping to FPGAs

Since SNE has been designed to be implemented in a custom chip, many elements belong to Standard Cell libraries (i.e. memory slices). Furthermore, the architectural template described in Section 5.1 could be exploited only within a more complex infrastructure that is capable of communicating with the APB programming module and streamers that expose interfaces to an external memory system. To achieve the SNE to FPGA porting objective, the architecture has been reviewed and revised to let some elements be mapped by using typical FPGA slices and subsequently, in favour of rapid implementation, it has been integrated within the NEURAghe infrastructure, effectively replacing the Convolutional Engine (1.4.1). A specific board has been targeted for this implementation, namely the Avnet Ultra96-V2 equipped with the Xilinx XCZU3EG MPSoC.

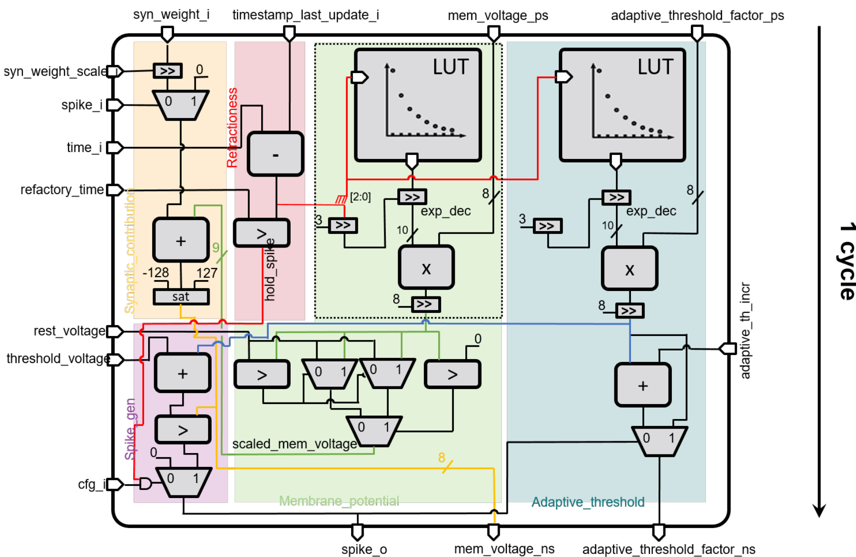


Figure 5.2: SNE Neuron data-path

It is a well-known issue that some arithmetical operations could be particularly wasteful in terms of resources if implemented with FPGAs Lookup Tables (LUTs), like multi-bit multiplications that, as seen in previous chapters, might be more effectively mapped to DSP slices instead. The first step towards an FPGA integration has been that of mapping memory elements to typical Block RAMs (BRAMs) of the FPGA and to map, as far as possible, neuron data-path multiply and accumulation operation to DSP slices.

Figure 5.2 shows the original neuron data-path configuration. It is possible to notice two main paths, one for the membrane potential update and the second

for the adaptive threshold, both containing a Lookup Table to approximate the potential decay and both are made by various multiply and add operations.

A preliminary synthesis of the simple neuron (with Xilinx Vivado) results in a resource utilization of 365 LUTs, i.e. none of the operations is mapped on DSP slices. For the architecture parameters mentioned above, by considering the available resources on XCZU3EG, resource utilization for neurons only is reported in Table 5.1a.

Table 5.1: SNE Neurons occupation before (a) and after (b) MAC ops mapped on DSPs

(a)

	DSP	BRAM	LUTs (logic)	LUTs (Mem)	Regs
Used	0	0	46720	0	0
Avail	360	432	70560	28800	141120
%	0	0	66.21	0	0

(b)

	DSP	BRAM	LUTs (logic)	LUTs (Mem)	Regs
Used	256	0	25984	0	0
Avail	360	432	70560	28800	141120
%	71.11	0	36.82	0	0

LUT utilization is about 71% of those available for the whole device, which may result in congestion problems when synthesizing the whole architecture. By modifying and instrumenting the HDL code to exploit DSP's datapath capabilities in terms of data-width and internal components (Pre-adder, Multiplier and ALU) it is possible to embed MAC operations on DSP slices, resulting in the Neuron datapath shown in Figure 5.3 and in the resource occupation shown in Table 5.1b. This solution reduces by about 50% the number of LUTs employed in the first one.

The second step concerns memory resources management. Internal Memory configuration is made up of several SRAM blocks used to store kernel weights, weights, both as status memory for events, as well as for streamers for context switch purposes. An explicit mapping of memory resources in FPGA integrated RAMB18 slices is necessary to avoid inefficiencies or resource waste of LUT and LUTRAM resources. A 2 KB RAMB18 Slice is the minimum instantiable memory resource for the target device which is sufficient to serve SNE units. In particular, as clusters need two independent accessible memory banks, working as event buffers, plus a private kernel memory bank, the total number of requested RAMB18 banks per

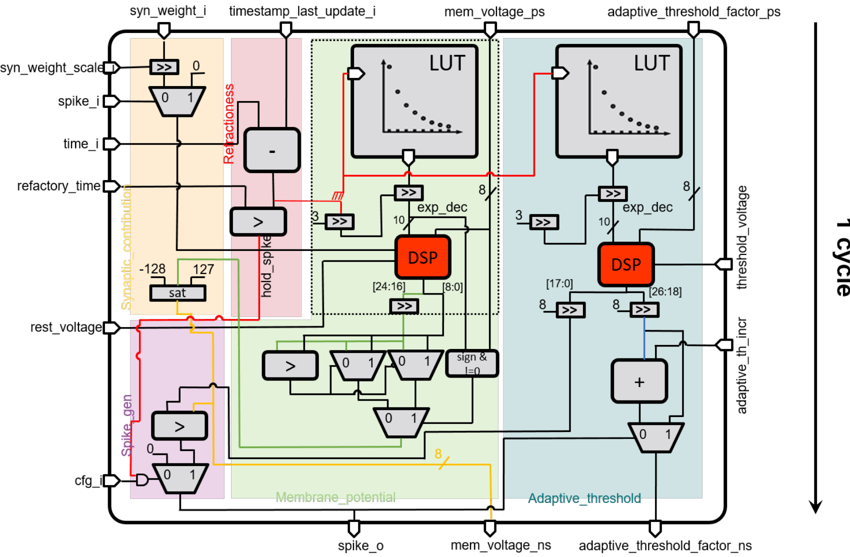


Figure 5.3: SNE Neuron Datapath: Ops to DSP mapping

SNE Slice unit are $16 \times 2 + 5$. Additionally, streamers need one independent bank each, for context switch purposes.
Despite the above-mentioned optimizations, an 8 (slices) $\times$ 16 (clusters) architecture is still too demanding with respect to the XCZU3EG available hardware resources. Table 5.2a (red) shows the overuse, especially of LUTs as Logic.

Table 5.2: SNE 8x16 (a) and 2x16 (b) architecture synthesis (Xilinx XCZU3EG)

(a)

	DSP	BRAM	LUTs (logic)	LUTs (Mem)	Regs
Used	256	298	106203	0	59340
Avail	360	432	70560	28800	141120
%	71.1	68.9	150.51	0	42

(b)

	DSP	BRAM	LUTs (logic)	LUTs (Mem)	Regs
Used	64	76	48238	0	26867
Avail	360	432	70560	28800	141120
%	17.7	17.6	68.3	0	19

Scaling the Slice parameter to 2 leads to more suitable resource utilization, as

shown in Table 5.2b, leaving the space for the infrastructure needed to complete the integration.

5.2.1 Exploiting NEURAghe infrastructure

Figure 5.4 shows the resulting architectural template after SNE integration within the NEURAghe infrastructure. A comparison with the base architecture depicted

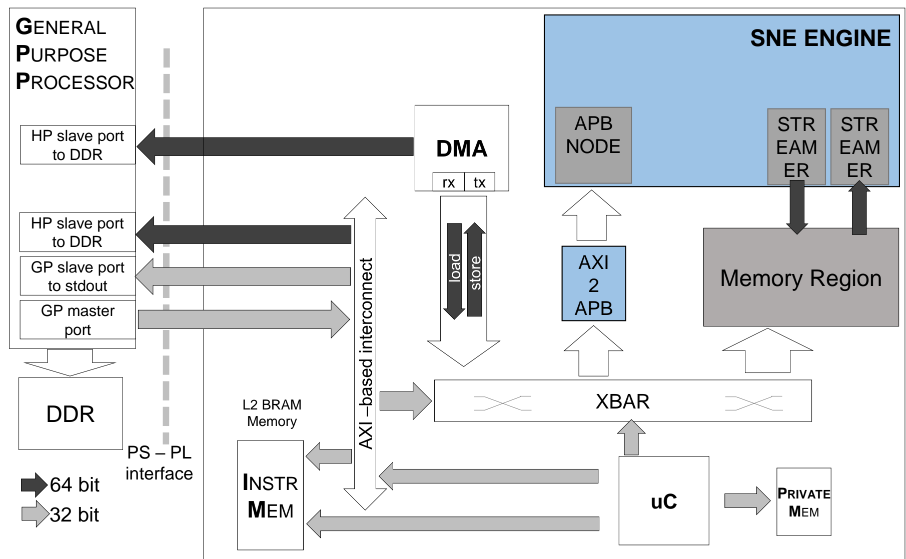


Figure 5.4: SNE integration within the NEURAghe infrastructure

in Figure 1.4 makes it possible to notice the substantial reuse of all the accelerator interface components along with some inter-PL IPs. In particular, regarding the re-configurable logic area, the obtained version maintains the micro-controller soft-core along with its private data and instruction memory, the AXI bus interconnect, a memory region and a DMA. Meanwhile the PS-PL port interface and the software stack have been entirely kept intact, apart for some slight enhancement to the accelerator offloading APIs to support SNE configuration and data transfer.

Thanks to the infrastructure flexibility it has been possible to do the integration in a relatively simple way resulting in an SNN accelerator where SNE could be seen as an external IP selectable at design time.

Streamer interfaces can easily communicate through its memory ports, directly connected to the memory region, to fetch DMA pre-loaded weight and event words during the inference phase. On the other hand, the APB node that exposes an APB protocol interface port needs a conversion logic from a peripheral protocol. APB configuration and scheduling parameters come directly from the programmable

soft-core running an SNE personalized middleware, which, as already discussed (Section 1.4), is programmed from the GPP and is in charge to handle synchronization among PL IPs.

5.2.2 Implementation Results

The architecture depicted in Section 5.2.1 has been synthesised using the Vivado Design Suite resulting in the resource occupation shown in Table 5.3a, reaching the maximum operational frequency of 60 MHz.

Table 5.3: SNN accelerator architecture resource occupation (XCZU3EG): without (60 MHz) and with (85 MHz) critical path optimization

(a)

	DSP	BRAM	LUTs (logic)	LUTs (Mem)	Regs
Used	64	172	63297	123	32658
Avail	360	432	70560	28800	141120
%	17.7	39.8	89.7	0.42	23.1

(b)

	DSP	BRAM	LUTs (logic)	LUTs (Mem)	Regs
Used	64	172	63346	123	33293
Avail	360	432	70560	28800	141120
%	17.7	39.8	89.8	0.42	23.6

As can be seen, despite the already described optimizations of the SNE, the overall LUT utilization is high (about 90%). This is partly due to the inherent accelerator dimension and partly to the specific SNE logic which in some cases must be completely combinatory (i.e. neuron data-path), resulting in architectural congestion that affects the critical path and limits the operating frequency. In order to overcome this issue, where feasible, some pipeline stages have been added to cut critical paths around neuron data-path, making it possible to reach 85 MHz, with a very slight LUT usage increment (Table 5.3b).

5.3 Experimental Results

As already stated, in the literature, there are other examples of SNN inference FPGA accelerators. The majority of them rely on MNIST data-set [67] to test their network architecture. Usually, they operate a conversion between a pixel-based 28×28 input frame to an event-based one, by mapping each pixel to a spike train with an activity rate that depends on pixel intensity with different methods ([77], [57], [114], [83]). Some of those implement a Multi-Layer Perceptron (MLP) like network, with few hidden fully connected layers aiming to maximise the fan-in/fan-out characteristics ([61], [30], [41], [40]) and, in some cases, by deploying the entire network model within the target FPGAs. The work of Irmak et al. [52] rely on Dynamic Partial Reconfiguration (DPR) to adapt an FPGA-based accelerator to work both on CNN and SNN by mean of different types of Processing Elements (PEs). The SNE PE has been tested on MNIST with a four fully connected layers architecture totally implemented into the FPGA. Also in the work of Corradi et al. [22] authors implement different SNN fully connected network topologies. In particular, they are designed for the sensory fusion cropland classification task on top of a Xilinx XCZU4EV and XCZU9EG MPSoCs. Kiselev et al. proposed an evolution of Minutaur [77] called n-minitaur [64] which is able to receive and process real-time spikes from up to three event-based sensors. It implements fully-connected networks on a Spartan-6 FPGA which has been tested on top of the MNIST dataset where generate spike trains depend on pixel intensity. Another approach is given by the work of Wang et al. [106] in which is shown the implementation of an hardware architecture for simulating large-scale and structurally connected spiking neural networks using simple LIF neurons trying to achieve the human brain scale and thus by relying on an Altera Stratix V FPGA, comprising 100 million neurons.

The miscellaneous context makes it not easy to carry out a fair comparison. Moreover, the SNN performance depends on various parameters such as the considered timing window (or encoding window for converted data sets), the spike activity, which in turn depends on the network's training parameters.

In [30] the authors propose a convolutional SNN architecture fully deployed in the Xilinx XCZU9EG FPGA, synthesised with the Vivado High-Level Synthesis (HLS) Software, reaching a maximum operating frequency of $125MHz$. Network characteristics are those shown in Table 5.4.

Table 5.4: [30] convolutional SNN architecture

input	conv	pool	conv	pool	fc	fc
28x28	32x3x3	32x2x2	32x3x3	32x2x2	256	10

The authors test the design with the MNIST dataset by applying their own encoding model with an encoding window $T_e = 10$. Trained parameters are of 16-bit fixed-point type. They state that the classification task has a latency of $7.53ms$ and the architecture is capable to process up to 2124 MNIST frame per second.

With the aim to compare the work of Fang et al. [30] with this work, we evaluate the execution time of the network in Table 5.4 with a growing global average neuron activity rate, by assuming to use the same encoding window of $T_e = 10$. Results are shown in Figure 5.5

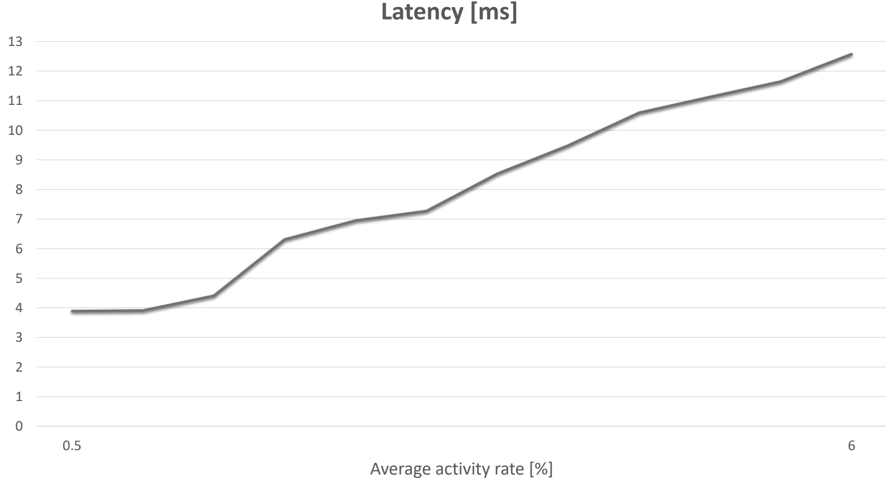


Figure 5.5: SNE latency trend with respect to the activity rate

As expected latency grows almost linearly with the neuron activity rate. From previous experiment on NMNIST a typical layer average activity rate, is 5%. Considering this value throughout all layers, the FPGA ported SNE architecture achieves a latency of $11.3ms$.

Table 5.5: A quantitative comparison between this work and [30].

	[30]	This work
frame rate	2124	88.49
frequency [MHz]	125	85
frame rate/dsp	1.184	1.383

Table 5.5 shows a quantitative comparison between the referenced work and this work. In particular, as authors in [30] declare a DSP utilization of 71.2% on the XCZU9EG MPSoC featuring a total of 2520 DSP slices, the ratio between the frame rate and the DSP count is 1.184. On the other hand, in this work, the target

device is a XCZU3EG MPSoC with an utilization of 64 to 360 DSP slices resulting in a frame rate to DSP ratio of 1.384.

Although the work in [30] offers generally better performance, implementing all the network architecture in a bigger device, the advantage in using SNE relies on its flexibility towards different layer characteristics that do not require an FPGA reconfiguration while changing the network topology.

6 | Conclusions

Convolutional Neural Networks lead Artificial Neural Networks algorithms when it comes to dealing with Computer Vision tasks such as, for example, image recognition, object detection or video frame classification, among others.

Their widespread diffusion during the years has brought researchers to question the opportunity to declinate their primary mission towards different use cases, resulting in the development of different approaches and devices.

This is the case, among others, of Temporal Convolutional Networks, LightWeight CNNs and Spiking Neural Networks which have also carried with them different ways to deal with data with respect to the classical ones, such as mono-dimensional management, event-driven processing and (specifically concerning the convolutional operator) various kernel sizes and dilation rate, depthwise separable convolution and its derived Mobile Bottleneck layer.

Moreover, motivated by the computational effort required to execute these kinds of networks, in parallel with the evolution of the algorithms, there has been also a growing interest in the deployment of specific hardware capable to accelerate their inference task, especially when an efficient, near-sensor data processing is required, as in the case of the edge computing field.

To cope with these needs, one of the most adopted solutions has been exploiting the capabilities of modern Multiprocessor Systems on-chip (MPSoCs) equipped with a general-purpose processor and FPGAs, among which those that belong to the Xilinx Zynq-7000 and Zynq Ultrascale+ families represent one of the most remarkable examples. This is the case of NEURAghe, a CNN inference accelerator whose processing model relies on the synergy between the ARM-based cores and a Convolution Specific Processor equipped with a programmable soft-core implemented in the reconfigurable logic.

In such a context the objective of this Ph.D. thesis has been to explore how a flexible adaptation of the edge-oriented FPGA-based acceleration task could be carried out with respect to the above-cited CNN related networks starting from the CNN inference accelerator NEURAghe.

First of all, we tested the flexibility of this accelerator leveraging its tunable parameters like the MAC Matrix size, the number of CSPs and the data precision.

This exploration leads to a variety of configurations that may be used in different use-cases to fit in different target architectures. This offers different trade-off optimization scenarios with respect to performance, cost, and power consumption.

Subsequently, following the trend that reconsiders the common association between Recurrent Neural Networks and sequence modelling tasks, in favour of a convolutional-based computation paradigm while dealing with time sequences [13], we presented the improvements made to the NEURAghe accelerator, in supporting Temporal Convolutional Networks.

The new features include the capability of supporting arbitrary *kernel_size*, *dilation rate* and *stride values* without overhead and a methodology for the TCN scheduling exploiting its sequence-based structure. We show the TCN data transfer management and performance of two computational paradigm approaches trading latency for throughput. This method has been applied to three use-case networks and uses two different SoCs as a target. Results with sample batching translate in up to 0.96, 0.86 and 0.57 of efficiency on the three use-cases, with respect to the peak achievable by each configuration. In the two use-cases with real-time requirements, adequate sample batching can be used to achieve sufficient throughput to timely process all input samples. Comparing the execution of the use-cases on a Cortex A53 quad-core and on an NVIDIA Maxwell GPU, it is possible to notice an execution time reduction and a power efficiency improvement respectively. Furthermore, the compatibility evaluation toward classical state-of-the-art CNNs shows 40% improvement when targeting irregular patterns.

Chapter 4 shows an analysis with respect to possible hardware solutions aiming to adapt the architectural template to the depthwise separable convolution and MBConv support without introducing excessive changes. The poor efficiency achievable pushed for a change in the computational paradigm while dealing with these kinds of operators and a Convolution Engine redesign which results in a design-time selectable IP capable to offer specific support to mobile network operators and layers, leading to a notable performance improvement highlighted comparing executions over MobileNetV2 MBConv layers.

Finally, from the collaboration with ETH Zurich, we presented a Spiking Neural Network accelerator FPGA integration. After an in-depth study of their ASIC targeting Spiking Neural Engine, SNE, we tackled the problem of conversion from standard cells to FPGA typical slices such as BRAMs and DSPs. This led also to a better resource utilization that, as a consequence, has requested for an architecture scaling to better exploit resources made available by the chosen Zynq Ultrascale+ board. We then apply the NEURAghe architectural template to exploit its peculiarity in the ARM/soft-core cooperation resulting in a topology agnostic FPGA-based SNN inference accelerator. The converted SNE IP could be quite easily integrated and thought of as an external engine to be chosen at design time

when it comes to dealing with SNNs. Despite difficulties for a fair SoA comparison, as performance for these kinds of networks are layer activity rate constrained, the referenced work results in general better performances, considering its greater operating frequency enabled by the bigger MPSoC targeted, but SNE relies on better flexibility exploitation towards different layer characteristics without requiring an FPGA reconfiguration.

The work described in this Ph.D. thesis produced the following publications:

- P. Meloni et al., "Exploring NEURAghe: A Customizable Template for APSoC-Based CNN Inference at the Edge," in IEEE Embedded Systems Letters, vol. 12, no. 2, pp. 62-65, June 2020, doi: 10.1109/LES.2019.2947312.
- M. Carreras et al., "Optimizing Temporal Convolutional Network Inference on FPGA-Based Accelerators," in IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 10, no. 3, pp. 348-361, Sept. 2020, doi: 10.1109/JETCAS.2020.3014503.

Bibliography

[1] "https://developer.arm.com/architectures/system-architectures/amba". (Cited on page 69)

[2] Accelerate your ai-enabled edge solution with adaptive computing. "https://www.xilinx.com/publications/ebooks/introducing-adaptive-system-on-modules.pdf". (Cited on page 2)

[3] hls4ml documentation. "https://fastmachinelearning.org/hls4ml/status.html". (Cited on page 8)

[4] Jetson modules technical specification comparison. "https://developer.nvidia.com/embedded/jetson-modules". (Cited on page 2)

[5] Xilinx Deep Learning Processing Unit. https://www.xilinx.com/products/intellectual-property/dpu.html. (Cited on page 9)

[6] Xilinx Vitis AI development environment. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html. (Cited on page 9)

[7] Zynq DPU v3.2. https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_2/pg338-dpu.pdf. (Cited on page 10)

[8] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015. (Cited on page 11)

[9] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. Hyperdrive: A multi-chip systolically scalable binary-weight cnn inference engine. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):309–322, 2019. (Cited on page 1)

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014. (Cited on page 28)

[11] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019. (Cited on page 50)

[12] Lin Bai, Yiming Zhao, and Xinming Huang. A cnn accelerator on fpga using depthwise separable convolution, 2018. (Cited on page 10)

[13] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*, 2018. (Cited on pages 2, 26, 28, 29, and 80)

[14] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994. (Cited on pages 2 and 28)

[15] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. Finn-¡i¿r¡/i¿: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), dec 2018. (Cited on page 8)

[16] Marco Carreras, Gianfranco Deriu, Luigi Raffo, Luca Benini, and Paolo Meloni. Optimizing temporal convolutional network inference on fpga-based accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 10(3):348–361, 2020. (Cited on page 10)

[17] Y. Chen, T. Yang, J. Emer, and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019. (Cited on page 1)

[18] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014. (Cited on page 26)

[19] S. Choi, K. Bong, D. Han, and H. Yoo. Cnnp-v2: A memory-centric architecture for low-power cnn processor on domain-specific mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(4):598–611, 2019. (Cited on page 1)

[20] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016. (Cited on page 54)

[21] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. (Cited on page 28)

[22] Federico Corradi, Guido Adriaans, and Sander Stuijk. Gyro: A digital spiking neural network architecture for multi-sensory data analytics. In *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, DroneSE and RAPIDO '21, page 9–15, New York, NY, USA, 2021. Association for Computing Machinery. (Cited on page 76)

[23] M. Courbariaux, Y. Bengio, and J. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3105–3113, 2015. (Cited on page 9)

[24] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *CoRR*, abs/1612.08083, 2016. (Cited on page 26)

[25] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018. (Cited on page 11)

[26] Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, Paolo Zambotti, Manuj Ayodhyawasi, Harvinder Singh, and Nalin Aggarwal. 14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 238–239, 2017. (Cited on page 2)

[27] Pierre Dutilleux. An implementation of the "algorithme à trous" to compute the wavelet transform. In *Wavelets*, pages 298–304. Springer, 1990. (Cited on page 29)

[28] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990. (Cited on page 28)

[29] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip C. Harris, Jeffrey D. Krupa, Dylan S. Rankin, Manuel Blanco Valentin, Josiah D. Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier M. Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. *CoRR*, abs/2103.05579, 2021. (Cited on page 8)

[30] Haowen Fang, Zaidao Mei, Amar Shrestha, Ziyi Zhao, Yilan Li, and Qinru Qiu. Encoding, model, and architecture: Systematic optimization for spiking neural network in fpgas. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020. (Cited on pages ix, 12, 76, 77, and 78)

[31] Yao Fu, Ephrem Wu, and Ashish Sirasao. 8-bit dot-product acceleration. *Xilinx Inc.: San Jose, CA, USA*, 2017. (Cited on pages 20 and 24)

[32] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, 2014. (Cited on page 11)

[33] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, May 2017. (Cited on pages 8 and 9)

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016. (Cited on page 2)

[35] Sebastian Goodfellow, Andrew Goodwin, Danny Eytan, Robert Greer, Mjaye Mazwi, and Peter Laussen. Towards understanding ecg rhythm classification using convolutional neural networks and attention mappings. 08 2018. (Cited on pages vii, ix, 26, 41, 43, 44, and 45)

[36] Alex Graves. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*, pages 5–13. Springer, 2012. (Cited on page 28)

[37] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013. (Cited on page 28)

[38] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018. (Cited on page 25)

[39] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [dl] a survey of fpga-based neural network inference accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):1–26, 2019. (Cited on page 7)

[40] Shikhar Gupta, Arpan Vyas, and Gaurav Trivedi. Fpga implementation of simplified spiking neural network. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4. IEEE, 2020. (Cited on pages 12 and 76)

[41] Jianhui Han, Zhaolin Li, Weimin Zheng, and Youhui Zhang. Hardware implementation of spiking neural networks on fpga. *Tsinghua Science and Technology*, 25(4):479–486, 2020. (Cited on pages 12 and 76)

[42] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014. (Cited on pages 1 and 26)

[43] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. (Cited on pages 1 and 26)

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. (Cited on page 41)

[45] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In *Advances in neural information processing systems*, pages 190–198, 2013. (Cited on page 28)

[46] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. (Cited on page 26)

[47] Matthias Holschneider, Richard Kronland-Martinet, Jean Morlet, and Ph Tchamitchian. A real-time algorithm for signal analysis with the help of the wavelet transform. In *Wavelets*, pages 286–297. Springer, 1990. (Cited on page 29)

[48] 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. *CoRR*, abs/1704.04861, 2017. (Cited on pages 1, 10, 54, and 56)

[49] Shehzeen Hussain, Mojan Javaheripi, Paarth Neekhara, Ryan Kastner, and Farinaz Koushanfar. Fastwave: Accelerating autoregressive convolutional neural networks on fpga. *arXiv preprint arXiv:2002.04971*, 2020. (Cited on page 52)

[50] Taras Iakymchuk, Alfredo Rosado-Muñoz, Juan F Guerrero-Martínez, Manuel Bataller-Mompeán, and Jose V Francés-Víllora. Simplified spiking neural network architecture and stdp learning algorithm applied to image classification. *EURASIP Journal on Image and Video Processing*, 2015(1):1–11, 2015. (Cited on page 12)

[51] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size. *CoRR*, abs/1602.07360, 2016. (Cited on pages 10 and 54)

[52] Hasan Irmak, Federico Corradi, Paul Detterer, Nikolaos Alachiotis, and Daniel Ziener. A dynamic reconfigurable architecture for hybrid spiking and convolutional fpga-based neural network designs. *Journal of Low Power Electronics and Applications*, 11(3), 2021. (Cited on page 76)

[53] E.M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, 2003. (Cited on pages 12 and 67)

[54] E.M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004. (Cited on pages 2 and 67)

[55] Yancao Jiang, Jie Ren, Xiang Xie, and Chun Zhang. Hardware implementation of depthwise separable convolution neural network. In *2020 IEEE 15th International Conference on Solid-State Integrated Circuit Technology (ICSICT)*, pages 1–3, 2020. (Cited on page 10)

[56] Xiping Ju, Biao Fang, Rui Yan, Xiaoliang Xu, and Huajin Tang. An fpga implementation of deep spiking neural networks for low-power and fast classification. *Neural Computation*, 32(1):182–204, 2020. (Cited on pages 12 and 67)

[57] Jacques Kaiser, Hesham Mostafa, and Emre Neftci. Synaptic plasticity dynamics for deep continuous local learning (decolle). *arXiv preprint arXiv:1811.10766*, 2018. (Cited on page 76)

[58] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016. (Cited on page 26)

[59] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014. (Cited on page 26)

[60] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019. (Cited on page 1)

[61] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. S2n2: A fpga accelerator for streaming spiking neural networks. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 194–205, 2021. (Cited on pages 12 and 76)

[62] Tae Soo Kim and Austin Reiter. Interpretable 3d human action analysis with temporal convolutional networks. *CoRR*, abs/1704.04516, 2017. (Cited on pages vii, ix, 26, 41, 46, and 47)

[63] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014. (Cited on page 26)

[64] Ilya Kiselev, Daniel Neil, and Shih-Chii Liu. Event-driven deep neural network hardware system for sensor fusion. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2495–2498, 2016. (Cited on page 76)

[65] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. (Cited on pages 1 and 26)

[66] Colin Lea, Michael D Flynn, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks for action segmentation and detection. In *proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 156–165, 2017. (Cited on pages 26 and 29)

[67] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. (Cited on page 76)

[68] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016. (Cited on page 67)

[69] Jiawen Liao, Liangwei Cai, Yuan Xu, and Minya He. Design of accelerator for mobilenet convolutional neural network based on fpga. In *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, volume 1, pages 1392–1396. IEEE, 2019. (Cited on page 10)

[70] Y. Ma, Y. Cao, S. Vrudhula, and J. s. Seo. An automatic rtl compiler for high-throughput fpga implementation of diverse deep convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2017. (Cited on pages 7, 8, and 9)

[71] Lukas Martak, Marius Sajgalik, and Wanda Benesova. Polyphonic note transcription of time-domain audio signal with deep wavenet architecture. pages 1–5, 06 2018. (Cited on pages vii, 41, 48, and 49)

[72] Alfio Di Mauro, Francesco Conti, Pasquale Davide Schiavone, Davide Rossi, and Luca Benini. Always-on 674 μw@4gop/s error resilient binary neural networks with aggressive sram voltage scaling on a 22-nm iot end-node. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(11):3905–3918, 2020. (Cited on page 2)

[73] Paolo Meloni, Alessandro Capotondi, Gianfranco Deriu, Michele Brian, Francesco Conti, Davide Rossi, Luigi Raffo, and Luca Benini. Neura¡span class="smallcaps smallercapital"¿ghe¡/span¿: Exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), dec 2018. (Cited on pages i, 9, 13, 14, 16, 26, and 30)

[74] Paolo Meloni, Daniela Loi, Gianfranco Deriu, Marco Carreras, Francesco Conti, Alessandro Capotondi, and Davide Rossi. Exploring neuraghe: A customizable template for apsoc-based cnn inference at the edge. *IEEE Embedded Systems Letters*, PP:1–1, 10 2019. (Cited on page 9)

[75] Sparsh Mittal. A survey of fpga-based accelerators for convolutional neural networks. *Neural computing and applications*, pages 1–31, 2018. (Cited on page 1)

[76] Simon W. Moore, Paul J. Fox, Steven J.T. Marsh, A. Theodore Markettos, and Alan Mujumdar. Bluehive - a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 133–140, 2012. (Cited on page 12)

[77] Daniel Neil and Shih-Chii Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628, 2014. (Cited on pages 12 and 76)

[78] Priyadarshini Panda and Kaushik Roy. Unsupervised regenerative learning of hierarchical features in spiking deep networks for object recognition. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 299–306, 2016. (Cited on page 67)

[79] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013. (Cited on page 28)

[80] A. Prost-Boucle, A. Bourge, F. Petrot, H. Alemdar, N. Caldwell, and V. Leroy. Scalable high-performance architecture for convolutional ternary neural networks on fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, Sept 2017. (Cited on page 9)

[81] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM. (Cited on pages 8 and 9)

[82] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong. Pir-dsp: An fpga dsp block architecture for multi-precision deep neural networks. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 35–44, 2019. (Cited on page 9)

[83] Bodo Rueckauer, Iulia-Alexandra Lungu, Yuhuang Hu, Michael Pfeiffer, and Shih-Chii Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, 11:682, 2017. (Cited on page 76)

[84] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018. (Cited on pages 54 and 56)

[85] Cicero D Santos and Bianca Zadrozny. Learning character-level representations for part-of-speech tagging. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1818–1826, 2014. (Cited on page 26)

[86] Matteo Antonio Scrugli, Daniela Loi, Luigi Raffo, and Paolo Meloni. A runtime-adaptive cognitive iot node for healthcare monitoring. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, page 350–357, New York, NY, USA, 2019. Association for Computing Machinery. (Cited on page 2)

[87] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in Neuroscience*, 13:95, 2019. (Cited on page 67)

[88] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. (Cited on page 25)

[89] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016. (Cited on page 52)

[90] Mark J Shensa. The discrete wavelet transform: wedding the a trous and mallat algorithms. *IEEE Transactions on signal processing*, 40(10):2464–2482, 1992. (Cited on page 29)

[91] M.J. Shensa. The discrete wavelet transform: wedding the a trous and mallat algorithms. *IEEE Transactions on Signal Processing*, 40(10):2464–2482, 1992. (Cited on page 1)

[92] Laurent Sifre and Stéphane Mallat. Rigid-motion scattering for texture classification. *CoRR*, abs/1403.1687, 2014. (Cited on page 2)

[93] Harsh Srivastava and Kishor Sarawadekar. A depthwise separable convolution architecture for cnn accelerator. In *2020 IEEE Applied Signal Processing Conference (ASPCON)*, pages 1–5, 2020. (Cited on page 10)

[94] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *ICML*, 2011. (Cited on page 28)

[95] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014. (Cited on page 28)

[96] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014. (Cited on pages 1 and 26)

[97] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018. (Cited on page 54)

[98] Y. Umuroglu, N. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 65–74, New York, NY, USA, 2017. ACM. (Cited on pages 8, 9, and 12)

[99] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *SSW*, 125, 2016. (Cited on pages 26, 29, 41, and 48)

[100] S. I. Venieris and C. S. Bouganis. Latency-driven design for fpga-based convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2017. (Cited on pages 7 and 9)

[101] Stylianos I. Venieris and Christos-Savvas Bouganis. Latency-driven design for fpga-based convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2017. (Cited on page 25)

[102] Stylianos I Venieris and Christos-Savvas Bouganis. Latency-driven design for fpga-based convolutional neural networks. In *2017 27th International*

Conference on Field Programmable Logic and Applications (FPL), pages 1–8. IEEE, 2017. (Cited on page 52)

[103] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas. *IEEE Transactions on Neural Networks and Learning Systems*, 30(2):326–342, 2019. (Cited on page 25)

[104] Mário P Véstias. A survey of convolutional neural networks on edge with reconfigurable computing. *Algorithms*, 12(8):154, 2019. (Cited on page 1)

[105] Erwei Wang, James J Davis, Peter YK Cheung, and George Constantinides. Lutnet: Learning fpga configurations for highly efficient neural network inference. *IEEE Transactions on Computers*, 2020. (Cited on page 9)

[106] Runchun M. Wang, Chetan S. Thakur, and André van Schaik. An fpga-based massively parallel neuromorphic cortex simulator. *Frontiers in Neuroscience*, 12, 2018. (Cited on page 76)

[107] Pete Warden and Daniel Situnayake. *TinyML*. O'Reilly Media, Incorporated, 2019. (Cited on page 2)

[108] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. (Cited on page 28)

[109] J. Weston. Dialog-based Language Learning. *ArXiv:1604.06045*, April 2016. (Cited on page 1)

[110] J. Weston, S. Chopra, and A. Bordes. Memory Networks. *ArXiv:1410.3916*, October 2014. (Cited on page 1)

[111] Jason E Weston. Dialog-based language learning. In *Advances in Neural Information Processing Systems*, pages 829–837, 2016. (Cited on page 26)

[112] Di Wu, Yu Zhang, Xijie Jia, Lu Tian, Tianping Li, Lingzhi Sui, Dongliang Xie, and Yi Shan. A high-performance cnn processor based on fpga for mobilenets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 136–143. IEEE, 2019. (Cited on page 11)

[113] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015. Withdrawn. (Cited on pages 1 and 26)

[114] Yang Xu, Huajin Tang, Jinwei Xing, and Hongying Li. Spike trains encoding and threshold rescaling method for deep spiking neural networks. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–6, 2017. (Cited on page 76)

[115] Aaron R. Young, Mark E. Dean, James S. Plank, and Garrett S. Rose. A review of spiking neuromorphic hardware communication systems. *IEEE Access*, 7:135606–135620, 2019. (Cited on page 11)

[116] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015. (Cited on page 29)

[117] Xiaoyu Yu, Yuwei Wang, Jie Miao, Ephrem Wu, Heng Zhang, Yu Meng, Bo Zhang, Biao Min, Dewei Chen, and Jianlin Gao. A data-center fpga acceleration platform for convolutional neural networks. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 151–158. IEEE, 2019. (Cited on page 7)

[118] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. Light-opu: An fpga-based overlay processor for lightweight convolutional neural networks. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 122–132, New York, NY, USA, 2020. Association for Computing Machinery. (Cited on page 11)

[119] Matthew D. Zeiler, Graham W. Taylor, and Rob Fergus. Adaptive deconvolutional networks for mid and high level feature learning. In *2011 International Conference on Computer Vision*, pages 2018–2025, 2011. (Cited on page 1)

[120] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2016. (Cited on page 7)

[121] J. Zhang and N. Verma. An in-memory-computing dnn achieving 700 tops/w and 6 tops/mm2 in 130-nm cmos. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):358–366, 2019. (Cited on page 1)

[122] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, abs/1707.01083, 2017. (Cited on page 10)