Politecnico di Torino

Master degree in Electronic Engineering

Master Thesis

Design of an FPGA-based accelerator for CNNs based on the Winograd algorithm

SUPERVISOR
Prof. Maurizio Martina

ADVISORS
Prof. Claudio Passerone
Prof. Guido Masera
Dott. Pierpaolo Mori
Dott. Emanuele Valpreda

CANDIDATE
Alessandra Vignoli

April 18, 2023

Abstract

Convolutional Neural Networks (CNNs) are a particular kind of Neural Network (NNs) that compute the outputs by means of a convolution operation between a set of 3D inputs and a 4D tensor weight. They find application in many fields such as image classification and object detection but their high prediction accuracy comes at the cost of high computation, large memory demand and a long inference time. Many attempts have been made at identifying the best hardware support and at researching strategies to concurrently accelerate the inference of CNNs while also limiting hardware complexity and improving flexibility. FPGAs have lately been the platform of choice because they offer a good compromise between flexibility and energy efficiency; data quantization on 8 bits has shown to reduce computation complexity while maintaining an acceptable accuracy. and the number of required multiplications can be reduced thanks to computational transforms such as the `F(4,3)` Winograd Algorithm, that is able to reduce the multiplication demands of up to $4\times$ with filters of size (3×3). However, Winograd algorithm applied to 8-bit data leads to a strong accuracy degradation.

This work presents an FPGA-based hardware accelerator for data quantized on 8-bits that exploits the complex F(4,3) Winograd algorithm, which allows to significantly reduce the numerical error. Resorting to Karatsuba's algorithm it is possible to reduce the number of multiplications by $3.13\times$ with respect to the standard convolution. The design is able to support stride-1 convolution with kernel size up to (4×4) and stride-2 convolution with kernel size up to (8×8) and achieves this flexibility without the need of additional hardware to support standard convolution. The main computational unit is a Processing Element based on the `F(4,3)` complex Winograd algorithm which presents higher flexibility in terms of supported kernel sizes with respect to other state of the art designs thanks to a transformation matrices reuse protocol. The architecture presents two organization elements responsible for proper input and weight decomposition and that allow for flexible tiling, efficient memory access and activation reuse thanks to the introduction of a flexible input buffer. The computational unit presents four Processing Elements working in parallel. Loop unrolling and data reuse are also used to improve the overall performance and real time transformation of data allows to reduce the memory demands.

Hardware execution shows no time degradation with the respect to the stand-alone Processing Elements, while the accelerator output numerical error is significantly lower than the one reported for the standard `F(4,3)` Winograd algorithm.

Contents

Chapter 1

Organization

This document is structured in 8 chapters.

- The second chapter contains a brief introduction on CNNs, their structure and operation and an analysis of possible hardware supports
- The third chapter introduces the Winograd algorithm as well as other techniques commonly used to accelerate convolution
- The fourth chapter reports a summary of the works that represent the starting point of the development of the current project
- The fifth chapter carefully analyses the optimization techniques that have been implemented in the design of the accelerator
- The sixth chapter describes the structure and design principles of the design
- The seventh chapter presents the results of the accelerator characterization in terms of resources utilization and output error
- The eight chapter summarises the contributions of this work and proposes possible further developments of the project.

Chapter 2

Background

Convolutional Neural Networks belong to the larger field of Artificial Intelligence (AI). The term Artificial Intelligence was coined in the 1950s by John McCarthy, a computer scientist who defined it as the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do. In the context of AI, a sub-field is represented by Machine Learning, that has been defined by Arthur Samuel as the field of study that gives computers the ability to learn without being explicitly programmed [1].

The development of Machine Learning has had the aim to implement algorithms that are able to extract information from the data they are fed with. One of the possibilities that have been developed to achieve this goal are Neural Networks (NNs), that represent an attempt at creating an algorithm that tries to emulate some of the aspects with which we understand the human brain to operate and that often takes the name of "brain-inspired computation" [1].

2.1 Brain operation

The fundamental block of the brain is the neuron, which is believed to be the core of the computation. Inside the average human brain it is possible to find around 86 billion neurons. Each neuron is connected to the surrounding ones by means of multiple entering elements called *dendrites* and a leaving element called *axon*. Each axon is then split into multiple branches that connect with the dendrites of other neurons by means of *synapses*. A dendrite is activated when it receives an electric stimulus from an axon: when multiple dendrites are activated, the neuron receives multiple inputs that it is able to combine and manipulate into an output signal, that is again fed to the connected neurons through the axon. During this procedure, the synapses are able to scale the incoming signal by a factor called *weight*: what allows the brain to be able to learn and adjust its computations is the update of these scaling factors [1].

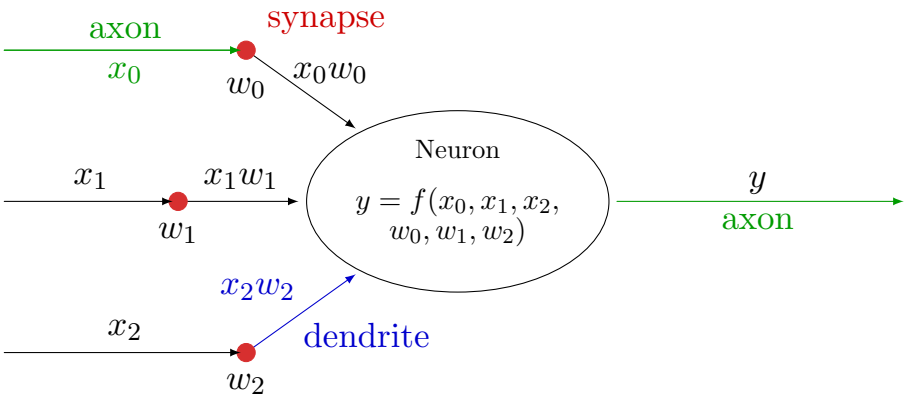


Figure 2.1: Graphical representation of the mathematical operation of a neuron with a number of inputs $N = 3$.

2.2 Neural Networks

Neural Networks are based on the key notion that the overall structure of the brain is not modified during the system operation. The world of NNs is composed of many different models, each characterized by its own structure. However, once the model is fixed the only parameters that need to change in order for one input to produce a different output are the weights.

NNs implementations try therefore to replicate a set-up similar to the one described above: the emulating system is made by a collection of fundamental units called neurons that apply a mathematical function. A graphical representation is shown in figure 2.1: each neuron receives N inputs $x_i,\ i \in [0, N-1]$ that are manipulated through N fixed scaling factors called weights $w_i,\ i \in [0, N-1]$ and then accumulated; finally, the accumulation result undergoes a non-linear operation to produce the neuron output. It is important to ensure the presence of some degree on non-linearity inside the network, because this is what makes it able to recognize complex data patterns: without this non-linearity in the computation the network would be only as powerful as one of its neurons [2].

Given M neurons $\xi_m,\ m \in [1, M]$ each accepting N inputs $x_n,\ n \in [1, N]$, given the weights $w_{n,m},\ n \in [1, N],\ m \in [1, M]$, and given the non linear function $f(\cdot)$, the set of neurons computes their outputs $y_m,\ m \in [1, M]$ as

$$y_m = f(\sum_{n=1}^{N} w_{n,m} \cdot x_n + b),\ m = 1, ..., M. \tag{2.1}$$

The term b is a correction factor called *bias*. A simple network example, with 5 neurons that take 3 inputs each, can be seen in figure 2.2. To create a parallelism with the human brain description, the inputs are usually referred to as *activations* [1].

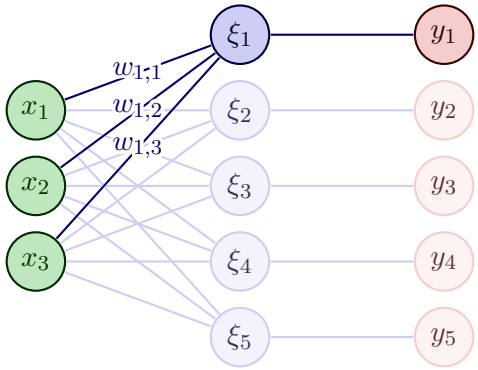


Figure 2.2: Example of a one-layer Neural Network with inputs $x_i,\ i \in [1, 3]$ and outputs $y_i,\ i \in [1, 5]$.

Training and inference

The use of NNs is characterized by two different steps, namely *training* and *inference*. During training, the network is asked to derive, or *learn*, the weight values that better allow the network to compute its output with a reasonably high prediction accuracy. Once the set of weights has been learned, the network is ready to be used for inference, when it is required correctly classify or label the input. As during the human brain operation, the internal connection of the network or the specific function that each neuron applies to its inputs do not change during neither the training process nor the inference time.

In a NN, neurons are usually organized in layers: each layer output activations propagate in the network as input activations to the following layer. The input and output layers are separated by a number of middle layers generally referred to as *hidden layers*. Different layers present a different number of neurons and inputs, and each neuron is characterized by its own set of weights and applies a specific function $f(\cdot)$.

2.2.1 Deep Neural Networks

Networks with more than three hidden layers are part of the domain of *deep learning* and are called Deep Neural Networks (DNNs). This particular kind of Neural Network is greatly used by modern AI applications since they are able to extract low- and high-level features from raw sensory data exceeding human accuracy [1].

DNNs organization

The many DNN layers can serve different purposes, which allow for their classification. In **Fully Connected layers**, neurons apply to the inputs the linear transformation already described in equation (2.1) and compute their output as the weighted sum of all the inputs.

Therefore given a layer with M neurons, N inputs x_n and the weights $w_{n,m}$, the M outputs y_m are computed as

$$y_m = \sum_{n=1}^{N} x_n \cdot w_{n,m},\ m = 1, ..., M \tag{2.2}$$

Since DNN layers can have a really large number of activations, these networks often require a large compute capability.

Non-linear layers are usually present after each FC layer and apply a non-linear manipulation to the data. The introduction of non-linearity is crucial during both training and inference. During training, it helps in reducing overfitting, which happens when the network memorizes the training data, but at the same time does not gain the ability to generalize its knowledge for the processing of new data [3]. During inference, it ensures the network with the ability to understand the complex relations that can be extracted from the input data [2]. Generally these layers define an activation function that allows the neurons to generate an output only if their input is compliant with some given requirements - e.g. their value is above a specific threshold. Common solutions for early NN implementations were the sigmoid and hyperbolic tangent nonlinear functions, shown in figure 2.3, that have however been shown not to be the optimal solution for DNNs, other than for specific applications [4]. A simpler and more popular solution for state-of-the-art DNNs is that of the `ReLU` (Rectified Linear Unit) function, defined as

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{2.3}$$

To improve the network accuracy while maintaining the implementation simplicity, recent works have also proposed slightly more complex functions derived from the ReLU one, among which leaky ReLU and exponential ReLU. A plot of the different ReLU activation functions can be seen in figure 2.3.

Normalization layers are used to adapt the distribution of the output activation of a layer to the following one. The use of these layers allows to better the accuracy and improve the training time [1]. The most common implementation is the so called *Batch normalization*, that manipulate the input distribution so to give it a mean $\mu = 0$ and standard deviation $\sigma = 1$.

DNNs applications

DNNs show improved prediction accuracy with respect to NN models that present a lower number of hidden layers. Today, DNNs find application in many fields such as image processing and classification, object detection [5], action recognition in video data [6], and speech recognition. However, this approach also leads to higher computation and memory demand due to the large number of hidden layers, with a consequent increase of the inference time [7], [8].

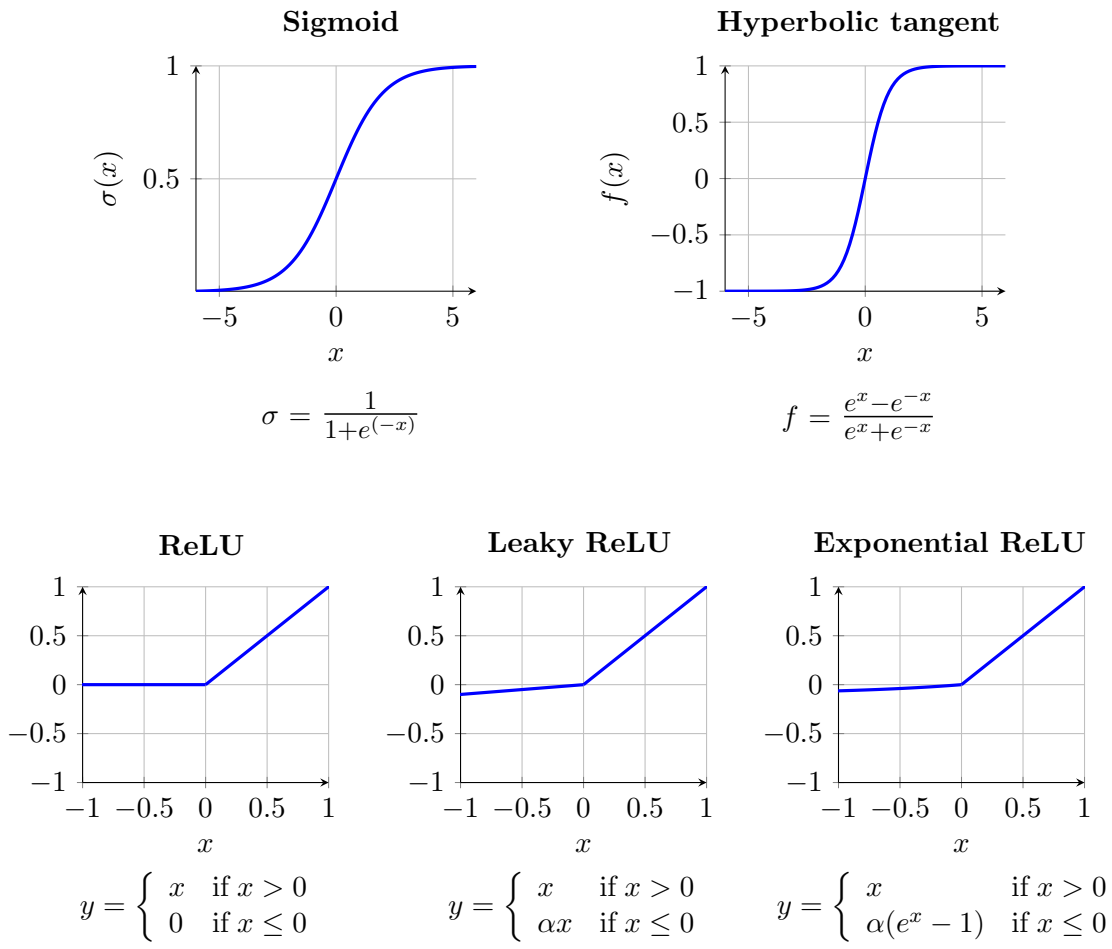
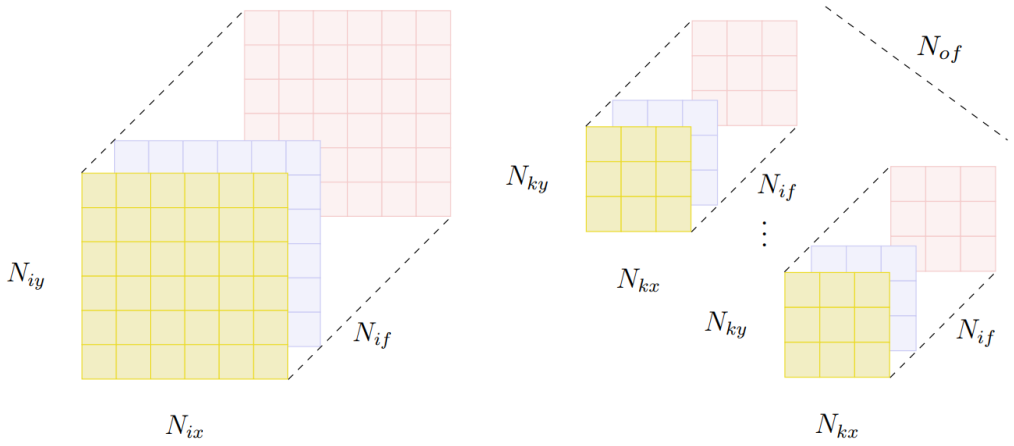


Figure 2.3: Sigmoid, Hyperbolic tangent and ReLU activation functions. The plots assume $\alpha = 0.1$.



(a) Spatial organization of a CNN layer input activation.

(b) Spatial organization of a CNN layer weights.

Figure 2.4: Graphic representation of the spatial organization of the input activations and weights in a CNN layer.

2.2.2 Convolutional Neural Networks

The computation of a DNN output requires a large amount of arithmetic and storage capability to the hardware of choice, due to the large amount of data involved. The structure complexity can however be reduced if the layer is implemented as a *sparsely connected layer*, where each output only depends on a spatial sub-set of the activations. This is the strategy exploited in Convolutional Neural Networks (CNNs), that are a particular kind of DNNs where most layers compute the output by means of a convolution operation between a set of 3D inputs and a 4D tensor of weights: these layers are called Convolutional (CONV) layers.

In this context the set of input data of a layer takes the name of *input feature map (ifmap)*: the activations are structured as a set of N_{if} 2D matrices called *channels*, each matrix with size $(N_{ix} \times N_{iy})$, as shown in figure 2.4a. As the name suggests, each $(N_{ix} \times N_{iy})$ feature map can be seen as a different representation of the same data, or a representation that highlights a specific *feature* of the input and that therefore contains different information. This is idea behind the characterization of an image pixels through their red, green and blue components.

The weights are instead organized as a bank of N_{of} 3D filters, each one with N_{if} channels and size $(N_{kx} \times N_{ky})$, as shown in figure 2.4b. In a CNN, the 2D filter is called *kernel* and each kernel is made by weight values trained to recognize a specific feature of the activations they are convolved with. The use of N_{of} 3D filters allows to extract N_{of} features from the same data. The same kernel can be reused to look for the same

feature in different spatial positions of the input feature map, by sliding the filter over the set of input activations. This makes the detection of the desired features invariant with respect to spatial transformations. In order to achieve this, the same weights values will be shared by a large number of a given layer neurons, which all receive a different sub-set of $N_{if} \times N_{kx} \times N_{ky}$ activations. Specifically, each layer neurons can be partitioned into N_{of} groups of $N_{ox} \cdot N_{oy}$ neurons, where all neurons of a specific group are characterized by the same $N_{kx} \cdot N_{ky} \cdot N_{if}$ weights and receive $N_{kx} \cdot N_{ky} \cdot N_{if}$ activations.

This *weight sharing paradigm*, coupled with the reduction of the number of inputs that contribute to the definition of an output value, is what allows CONV layers to reduce the computation complexity of DNNs. Moreover, weight sharing is the key characteristic that makes CNNs learning capabilities more efficient and effective with respect to other kinds of NNs.

Convolution output feature map

The convolution operation applies to the input data a linear transformation that gives the outputs a shape that is strongly linked with that of the inputs and filter. Indeed the set of output data takes the name of *output feature map (ofmap)* and replicates the same organization used for the inputs: the output activations are represented by a 3D matrix with size $(N_{of} \times N_{ox} \times N_{oy})$.

Simple convolution operation

In its simpler form, the convolution operation involves a one-channel ifmap and a 2D filter.

The computation depends on four main parameters:

1. the ifmap size $(N_{ix} \times N_{iy})$
2. the kernel size $(N_{kx} \times N_{ky})$, that also defines the size of the fixed-size ifmap window of data that contributes to the computation of a single output value;
3. the stride s, that specifies the distance in terms of ifmap values that the filter slides over between one window of data and the following one;
4. the zero padding p, that identifies the number of 0's that must be added at the boundary of the ifmap.

All these parameters contribute to the definition of the ofmap dimensions. Assuming the ifmap to have along both axis dimensions $i = N_{ix} = N_{iy}$, the kernel to have dimensions $r = N_{kx} = N_{ky}$, the convolution with stride s and padding p will give an ofmap with dimensions

$$N_{ox} = N_{oy} = \left\lfloor \frac{i + 2p - r}{s} \right\rfloor + 1 \tag{2.4}$$

Algorithm 1 Convolution operation

```
for oy ≤ N_iy do
    for ox ≤ N_ix do
        for of ≤ N_of do
            for if ≤ N_if do
                for y ≤ N_ky do
                    for x ≤ N_kx do
                        ofmap[of, oy, ox] = ifmap[if, s·oy + y, s·ox + x] * filter[of, if, y, x]
```

At each iteration of the convolution computation, the kernel will cover a $(r \times r)$ portion of the ifmap that takes the name of *window*. Each output value is computed through the multiplication between the kernel values and the ifmap values at the positions they overlap with and the accumulation of these products. Different output values can be computed by moving the kernel over the ifmap by a number of locations defined by the stride value. Each iteration of the algorithm computes one output values and require the convolution of the kernel with a different tile: therefore both the total number of iterations and number of tiles T are equal to $N_{ox} \cdot N_{oy}$. Figure 2.5 shows an example of a convolution operation with a layer with ifmap dimensions $i = 4$, kernel dimensions $r = 3$, stride $s = 1$ and padding $p = 1$.

Convolution operation

When the ifmap and kernel present a number of channels $N_{if} > 1$, each 2D kernel of the filter is convolved with a different ifmap channel and the layer output is given by the spatial accumulation of all these separate convolutions. When the weight tensor presents a number of filters $N_{of} > 1$, the convolution of each 3D filter with the full ifmap leads to the computation of one of the N_{of} output channels. Therefore denoting $D_{c,\alpha,\beta}$ the ifmap tile taken from channel $c \leq N_{if}$ and used for the computation of the output pixel in position $\alpha \in [1, N_{ox}]$ and $\beta \in [1, N_{oy}]$, and denoting $G_{k,c,u,v}$ a filter with parameters $k \leq N_{of}$, $c \leq N_{if}$, $u \leq N_{kx}$ and $v \leq N_{ky}$, the output of the CONV layer can be computed as:

$$Y_{k,\alpha,\beta} = \sum_{c=1}^{N_{if}} \sum_{v=1}^{N_{ky}} \sum_{u=1}^{N_{kx}} D_{c,x+u,y+v} \cdot G_{k,c,u,v}, \ \forall\ \alpha \leq N_{ox},\ \beta \leq N_{oy},\ k \leq N_{of}. \tag{2.5}$$

Algorithm 1 shows a possible implementation of the convolution.

From equation (2.5) it is possible to evaluate that the number of required multiply operations N_{multops} is

$$N_{\text{multops}} = N_{of} \cdot N_{oy} \cdot N_{ox} \cdot N_{if} \cdot N_{ky} \cdot N_{kx} \tag{2.6}$$

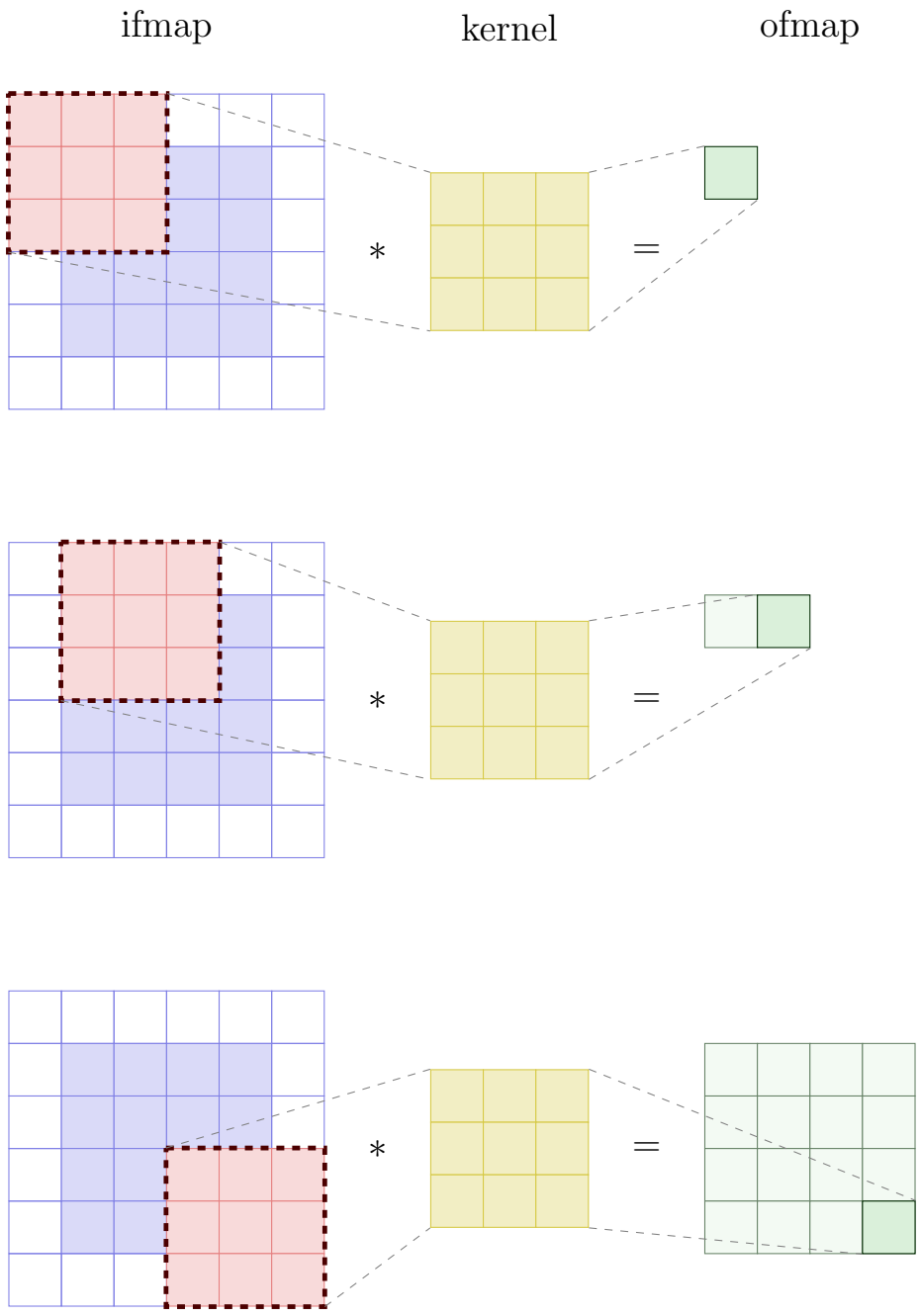


Figure 2.5: Example of convolution operation with a layer with ifmap dimension $i = 4$, kernel dimension $r = 3$, stride $s = 1$ and padding $p = 1$. The white cells in the ifmap represent the padding.

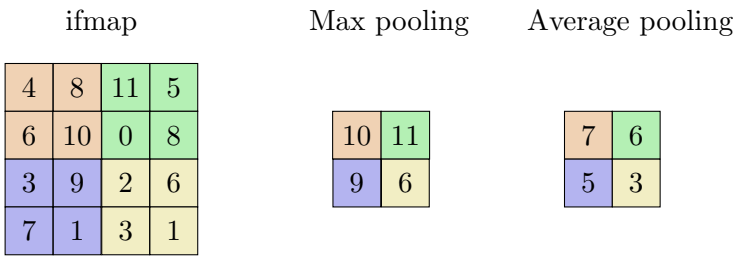


Figure 2.6: Example of Max pooling and Average pooling with a (2 x 2) filter and stride $s = 2$.

The N_{of} filters are used to extract different information from the same input data, therefore the N_{of} channels of the output activations can again be seen as the representation of a different *feature* of the same data, as they do for the ifmap.

Pooling layers

Alongside CONV layers, CNNs also present some FC layers at the end of the network, to perform classification tasks, as well as the other kind of layers that make up standard DNNs. In addition to those, the peculiar organization of CNNs input activations as matrices makes it possible to use pooling layers to reduce the input feature map spatial dimensions by combining multiple neighbouring values into a single one: pooling is therefore a down-sampling operation that as such decreases the resolution of the ifmap, but without modifying the number of channels.

Pooling layers, like CONV layers, are characterized by an $(r \times r)$ filter that slides over the ifmap according to a stride value s. Each iteration produces one output value following a specific pooling paradigm. Among the different pooling paradigms there are Max and Average pooling. **Max pooling** selects the maximum among the values of the current window tile; **Average pooling** instead computes the average. An example of a pooling layer output can be seen in figure 2.6.

Usually the value of s is chosen so that $s \geq r$, i.e. so that neighbouring tiles do not overlap: a larger stride value reduces the ifmap spatial resolution but simplifies the computation complexity of the following layers. It is interesting to note that performing a convolution with a stride greater than 1 can be seen as a form of pooling on its own, with the important difference that the convolution computation does not represent in this case a down-sampling operation but is again a search for features with no information loss.

2.3 Hardware accelerators for CNN inference

Both CNN training and CNN inference require many MAC operations and the use of a large number of parameters, but they present different requirements in terms of computation capability, resources utilization, and acceptable computation time. Multiple works focus on training optimization and research strategies to make it more efficient and robust [9], but they are outside the scope of this work, which focuses on CNN inference. CNN inference poses many limitations in term of acceptable results accuracy, computation speed and energy consumption, and these problems are enhanced when inference has to take place on embedded platforms and for real-time applications. The design of hardware accelerators for NN inference faces therefore many challenges since an accelerator should be able to provide a reasonable computing speed while also be compliant with the availability of area and resources and with the energy consumption limitations.

2.3.1 Inference requirements

A standard CNN is made of tens to hundreds of layers [10], each characterized by different filter shapes and stride values [1]. Ideally, a hardware accelerator should be able to process the full network, independently on the current layer specifications, on the same hardware support and with the lowest possible resources overhead. In other words, an efficient hardware accelerator should be able to grant an acceptable amount of flexibility in terms of supported layers.

Additionally, the design must take into account that each layer is characterized by its own set of weights, which need to be stored in a large enough memory that the accelerator should be able to access with an efficient memory access paradigm.

Finally, the computation requires a large number of MAC operations. The use of the available hardware resources must be optimized so to allow for a large throughput while also be compliant with the energy efficiency requirement.

To summarize, the design of a hardware accelerator should provide:

- flexibility to support different filter shapes and stride values;
- enough computation resources to allow for computation parallelization and high throughput;
- a DRAM large enough to store all the required layer parameters and weights, as well as the initial inputs, coupled with an efficient memory access paradigm; and the allocation of a reasonably deep on-chip memory to allow local storing of the current data
- minimal output prediction accuracy degradation due to errors inevitably introduced during the computation

2.3.2 Inference optimization techniques

Among the techniques that have been proposed to improve the computation complexity and the layer execution time while maintaining an acceptable accuracy there are quantization, tiling, loop unrolling and computation parallelization, data reuse, and the use of computational transforms.

Reduced precision

State-of-the-art NNs work with data stored in floating point format on 32 bits, which allow to have a larger dynamic range with respect to 32-bit integer or fixed point formats. Performing operations with float values requires however a larger amount of logic and computation time. Simplification of the computation can be achieved by either using floating point values with a lower bitwidth or exploiting fixed-point (fractional or integer) representation instead of the floating point one. However, to see reasonable improvements it is necessary to both apply a bitwidth reduction and use a fixed-point representation at the same time [1].

Data precision can be reduced by mapping the set of input values with a smaller set of quantization points. The new data requires a lower number of bits to be represented and therefore can help in lowering the computation complexity, as well as the computation time: recent studies have shown that the use of an 8-bit fixed-point representation instead of 32-bit floating point data reduces the energy consumption of MAC operations by 18.5× and the required area of 27.5× [1], [11], and increases the throughput by 4× [12], all without excessively reducing the output accuracy. Having the data on a lower number of bits also means a smaller memory space and lower energy consumption during memory access.

It is important to take into consideration that the data internal precision will still be larger than the nominal one, so to guarantee a low precision loss. After the convolution operation the output data precision must again be reduced to the original number of bits. The result still presents a reasonable accuracy even after the bitwidth reduction if weights and input activations have a distribution with mean $\mu = 0$, which shows the importance of batch normalization for the inputs [1].

Loop unrolling and computation parallelization

The majority of the computation in CNN inference concerns the multiply-and-accumulate (MAC) operations proper of convolutions. Because MAC operations can be easily parallelized, it is easy to improve the computation speed if a large enough number of hardware resources can be allocated for the computation. Loop unrolling can therefore increase the parallelization and subsequently the computation throughput.

Tiling and memory access optimization

Tiling is an efficient technique to reduce the number of memory accesses. It requires the allocation of on-chip buffers on which chunks of data can be efficiently loaded from the external memory if they are read from contiguous memory locations. Moreover, it can speed up the computation because once the data is on the on-chip memory it can be more readily retrieved.

Computational transforms

The demands of the convolution operation in terms of complexity and resources can be reduced thanks to computational transforms, such as the FFT, the Strassen algorithm [13] or the Winograd algorithm [14]: these computational transforms apply a data manipulation and, when applied to the activations or kernel, are able to effectively reduce the number of multiplications required to compute the convolution result.

Thanks to this reduction in the number of multiply operations, an accelerator can increase throughput and reduce the required computation resources. However as their name suggests, computational transforms perform a manipulation of the data that often leads to weights and activations bitwidth increase and to the introduction of redundant data. Any improvement is therefore achieved at the expense of both an increase in the memory demand and a more complex memory access pattern.

2.4 Hardware support

The choice of the correct hardware support is of crucial importance for a DNN hardware accelerator, because it must allow the allocation of an amount of resources compatible with the networks complexity in terms of number of parameters, number of MAC operations, as well as ensuring a low energy consumption despite the large number of operations. This is even more true when the design is to be used for inference of real-time applicationson edge devices, that present additional requirements in terms of acceptable computation speed and available area.

Different accelerator designs have been developed on all Graphical Processing Units (GPUs), Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). The solution of choice strongly depends on the requirements and limitations of the application and the target environment, and should be able to allow for parallelization of the computation, reduce the energy consumption, grant low latency, while also keeping in mind the high computation and memory demands.

GPUs

GPUs are the most common support of choice since it is able to provide large compute capability and memory bandwidth. This kind of architecture exploits a large number of

ALUs during the computation, often organized in such a way to exploit parallelization techniques to speed up the computation [1]. However GPUs have a large power consumption, which makes them not favourable for embedded, real-time applications [15], [16].

ASICs

ASIC are by definition specifically designed for the target application: ASIC-based hardware accelerators can therefore be optimized so to present a high throughput while also granting energy efficiency [17]. This is however achieved at the expense of lack of reconfigurability, since once the design is finalized it cannot be modified [16]. Additional drawbacks of this approach are the larger cost and the longer development time.

FPGAs

Field Programmable Gate Arrays are programmable logic devices that contain an array of Configurable Logic Blocks (CBLs) connected through multiple reconfigurable interconnections: this makes them easily programmable to target the specific task. FPGAs can offer high flexibility in terms of reconfigurability and are characterized by a lower design time with respect to ASICs.

FPGA-based accelerators can exploit the large number of available DSPs to strongly parallelize the computation and achieve low latency inference However, maximization of resources utilization is not always possible due to the limited on-chip memory available.

State-of-the-art CNN accelerators are increasingly often FPGA-based, because among the proposed supports FPGAs are the ones that grant the better compromise between performance, flexibility and energy efficiency [7].

2.4.1 High-Level Synthesis

High-Level Synthesis (HLS) has been lately often used for the design of hardware accelerators. The design desired behavior is usually described with a High-Level Language (HLL) such as C, System C, Python or Matlab, which allows for an easier simulation and debug of the design and enhances the portability of the code to different devices. Specific HLS tools, such as Vitis HLS for Xilinx platforms, than use HLS languages to translate HLL into an RTL description. During this process, HLS tools explore different computation parallelism or scheduling solutions as well as different resources allocation or sharing paradigms. The RTL synthesis can also be controlled by the designer through the specification of a set of instructions that the compiler is able to identify as constrains and not as part of the behavioral description: these constrains can control throughput optimization techniques such as loop unrolling and pipelining, can impose possible memory partition or can enforce limitations to the resource utilization. All these design choices obviously have an impact on the final energy consumption [18], [19].

HLS tools are becoming increasingly popular because they can efficiently explore the design space by analysing multiple possible implementation solutions without the need to modify a hand-coded, low-level design description and because they also provide, along the RTL, an analysis of the device timing and an estimation of the resource utilization, which allow for an easy comparison of the possible solutions [18].

Chapter 3

Introduction

This chapter illustrates the theoretical background that can help understand starting point of this project and the design problems that previous works have highlighted.

3.1 Winograd Algorithm

The Winograd algorithm is a minimal filtering algorithm that takes its name from Shmuel Winograd, who first introduced it in 1980 [20]. Minimal filtering algorithms allow to compute a specific number of outputs with the lower possible number of multiplications.

The use of this algorithm when performing a convolution operation allows to reduce the arithmetic complexity of the computation thanks to the transposition of activations and weights into a new domain called Winograd domain, where multiple outputs can be computed with a single element-wise matrix multiplication. When filters have a small kernel size (e.g. $r = 3$), this computational transform is favourable with respect to the conventional FFT convolution algorithm because of its lower memory requirements [14].

3.1.1 Generalities

The name Winograd algorithm refers to a class of minimal filtering algorithms identified by the name `F(m,r)`, where `r` identifies the size of the kernel and `m` the number of outputs that are to be computed together. In the case of 1D convolution, the use of this algorithm when performing a convolution operation with a filter with kernel size `r` allows to reduce the arithmetic complexity of the computation because it allows to compute the `m` outputs by performing $m + r - 1$ multiplication, instead of the $r \cdot m$ required by the standard convolution [14]. The multiplication reduction depends therefore on the filter and block of inputs sizes and increases with the amount of inputs processed together [1].

In the context of 2D convolution, the nomenclature `F(`$m_x \times m_y$`,`$r_x \times r_y$`)` refers to the particular Winograd algorithm that performs the convolution with a filter of size $(r_x \times r_y)$ and that computes $(m_x \times m_y)$ outputs at every iteration. In this case, the algorithm takes

as input matrices called *image tiles* of size $((m_x + r_x - 1) \times (m_y + r_y - 1))$ and the output computation requires $(m_x + r_x - 1) \cdot (m_y + r_y - 1)$ multiplications.

In the following we will assume to work with square filters where $r = N_{kx} = N_{ky}$ and the nomenclature `F(m,r)` will indicate the Winograd algorithm that performs the convolution with a filter of size $(r \times r)$ and that computes $(m \times m)$ outputs at every iteration. The algorithm takes as input image tiles of size $((m+r-1)\times(m+r-1))$ and computes the output through an element-wise matrix multiplication: therefore, it requires $(m+r-1)\cdot(m+r-1)$ multiplications (i.e. only one multiplication per element of the image tile) instead of $(r \cdot m)^2$ of the standard convolution. The global arithmetic complexity reduction Φ of Winograd convolution in the case of square tiles is

$$\Phi = \frac{m^2 r^2}{(m + r - 1)^2} \tag{3.1}$$

Winograd domain

The previous section stated that the algorithm is able to compute a block of $(m \times m)$ outputs through an element-wise matrix multiplication between a $(r \times r)$ filter and an $(n \times n)$ image where $n = m + r - 1$. This might seem counter-intuitive since the three matrices have different dimensions.

Indeed, in order to exploit the algorithm both the input and filter must be transformed into the Winograd domain. Both transformations can be performed thanks to a matrix multiplication with a transformation matrix that is specific of the `F(m,r)` algorithm that is being used. Once the data has been transformed, the two sets of data will have the same dimensions and the Winograd output can be computed with a simple element-wise matrix multiplication. Finally, the convolution outputs can be retrieved with an additional matrix multiplication which allows to transpose the output back into the original domain. A graphical representation of the algorithm operation can be found in figure 3.1. The whole procedure can be described in matrix form: assuming (with the same nomenclature used in [14]) g to be an $(r \times r)$ kernel with transformation matrix `G`, d to be an $(n \times n)$ image tile with matrix transformation `B` and `A` to be the output matrix transformation, the $(m \times m)$ outputs y can be computed as

$$y = A^T \left[[GgG^T] \odot [B^T dB] \right] A \tag{3.2}$$

where $\odot$ represents the element-wise matrix multiplication.

The algorithm transformation matrices are derived following the Cook-Toom algorithm and the value of their elements depends on the values chosen for different interpolation points [21]. The number of required interpolation points changes with the specific Winograd algorithm and their choice impacts the arithmetic complexity of the transformation itself. The interpolation points are considered good when they allow to generate transformation matrices that require only additions, subtractions and small shifts to transpose the data

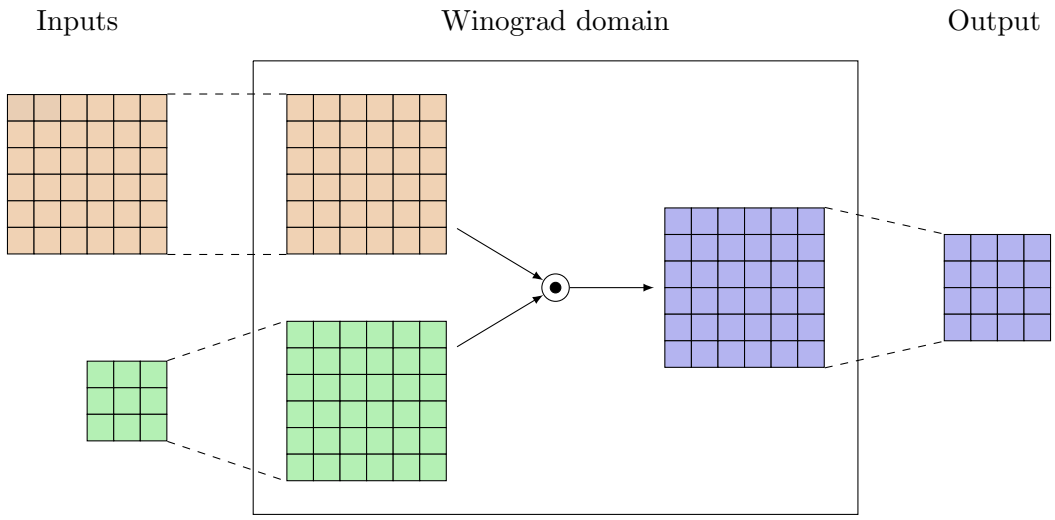


Figure 3.1: F(4,3) Winograd algorithm.

into the Winograd domain [22], but this is not possible for every chioce of the parameters `r` and `m`.

Eq. (3.2) clearly shows how the improvement evaluated through eq. (3.1) is achieved at the expense of the arithmetic complexity required to transform the data into the Winograd domain, which depends on the specific Winograd algorithm and on the choice of the interpolation point, increases with the image tile dimensions; and can have a strong impact on the final convolution computation complexity. Therefore while looking at eq. (3.1) one might think that the use of bigger image tiles would allow a larger computation complexity reduction, the transformation process makes the use of larger tiles globally inefficient.

Pipeline possibilities

Despite the problems highlighted in the previous section, figure 3.1 is able to show how through Winograd algorithm the convolution can be split into three steps, namely the inputs transformation, the Winograd-domain convolution, and the output transformation. The separation of the computation of these three steps allows to implement a pipelined architecture that as such can decrease the iteration latency. This represents one of the big advantages of the Winograd convolution.

Winograd iterations

To compute the convolution between a given layer and an $(r \times r)$ weight with the Winograd algorithm, multiple $(n \times n)$ image tiles must be generated from the input feature map. Since $(m \times m)$ outputs are computed from every image tile, neighbouring tiles must be taken by

Algorithm 2 Winograd convolution

```
Image tiles: T = ceil(Nix/m) * ceil(Niy/m)
Winograd Size: n = m + r - 1
Arry of image tiles: d[Nif,T][n,n]
Array of kernels: g[Nof,Nif][r,r]
Array of output tiles, Winograd domain: M[T][n,n]
Array of output tiles: d[Nof,T][m,m]

for of ≤ N_of do
    M[T][n,n] = 0
    for if ≤ N_if do
        % weight transformation into Winograd domain
        u = G*g[of,if]*G'
        for t ≤ T do
            % image tile transformation into Winograd domain
            v = B'*d[if,t]*B
            for y ≤ n do
                for x ≤ n do
                    % element − wise matrix multiplication
                    M[t][y,x] += u[y,x] * v[y,x]
                end for
            end for
        end for
    end for
    for t ≤ T do
        % output inverse transformation
        m = M[t]
        y[of,t] = A'*m*A
    end for
end
```

sliding the $(n \times n)$ window with a new stride value $s_W = m$. This means that every new tile will present $r - 1$ overlapping elements with the previous one. Considering an ifmap with dimensions N_{ix}, N_{iy} and N_{if}, the algorithm will therefore generate

$$\theta = \left\lceil \frac{N_{ix}}{m} \right\rceil \cdot \left\lceil \frac{N_{iy}}{m} \right\rceil \tag{3.3}$$

tiles for every input and output channel. The number of iterations of the algorithm is again equal to T. An example of a possible implementation for the solution of a CONV layer through the Winograd `F(m,r)` algorithm can be found in algorithm 2. This implementation

also highlights how the computation cost of the inverse transforms can be reduced if the different input channels are accumulated in the Winograd domain: the use of this strategy reduces the number of inverse transformations by a factor of $T \cdot N_{if} \times$.

3.1.2 Standard Winograd F(4,3) algorithm

The particular version of the algorithm that will be used as reference in the following is the Winograd `F(4,3)` algorithm. This variant of the algorithm has been largely analysed in the literature and is a popular way to accelerate the convolution of layers characterized by filter dimension of (3 x 3), that is the most common kind of filter in the ResNet18 model [23]. As can be understood from its definition, the `F(4,3)` algorithm can be used to compute the convolution between some input activations and a filter of size (3 x 3) and computes output tiles of dimension (4 x 4). This operation requires input image tiles of size (6 x 6) and 36 multiplications, whereas the standard convolution would require 144 operations: this particular form of the algorithm is therefore able to grant a $4x$ reduction for the number of multipliers.

These numbers however do not take into account the computation capability required by the transformation into the Winograd domain.

Transformation matrices

The derivation of the transformation matrices characteristic of `F(4,3)` algorithm requires five interpolation points: the best-known choice is $[0, 1, -1, 2, -2]$ and gives the transformation matrices [14]

$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \tag{3.4}$$

$$G = \begin{bmatrix} 1/4 & 0 & 0 \\ -1/6 & -1/6 & -1/6 \\ -1/6 & 1/6 & -1/6 \\ 1/24 & 1/12 & 1/6 \\ 1/24 & -1/12 & 1/6 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.5}$$

Model	Data format	Ops ratio	Maximum difference	Mean	Standard deviation
Winograd standard	Floating point	3.72	0.034		
Winograd standard	8-bit Quantized	3.72	137	7.23	28.14

Table 3.1: Analysis of the results of the Standard `F(4,3)` Winograd algorithm with respect to the standard convolution results, for both floating point and quantized data. The table reports the maximum difference between the analyzed Winograd models and the standard convolution and, when relevant, the error mean and standard deviation. The test layer has dimensions $N_{ix} = 54$, $N_{iy} = 54$, $N_{if} = 32$, $N_{of} = 32$ and is convolved with a (3×3) filter and stride 1.

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 0 \end{bmatrix} \tag{3.6}$$

Globally, the direct and inverse transformations require 328 constant floating point multiplications in addition to the 36 required by the Winograd convolution.

Despite being the best-derived algorithm, the filter transformation matrix contains the value 24 as the denominator of two of its elements, which is obviously not a power of 2's. Because of this, the transformation cannot be implemented through additions and shift operations and presents a large arithmetic complexity.

Convolution model

The impact of the transformation into and from the Winograd domain on the algorithm final output has been evaluated thanks to a Python script that models the convolution computation through the Winograd algorithm and compares these results with those of the standard convolution. As can be seen from the results reported in table 3.1, the output error, evaluated as the difference between the results of the standard convolution and those computed with the standard Winograd algorithm, is negligible. The error distribution is depicted in figure 3.2. The table also reports a MAC reduction that is lower than the theoretical one: this happens because the number of computed outputs $N_{ox,w}$ is always a multiple of $m = 4$: the value $N_{ox,w}$ might therefore be greater than N_{ox}. The same holds for $N_{oy,w}$ and N_{oy}.

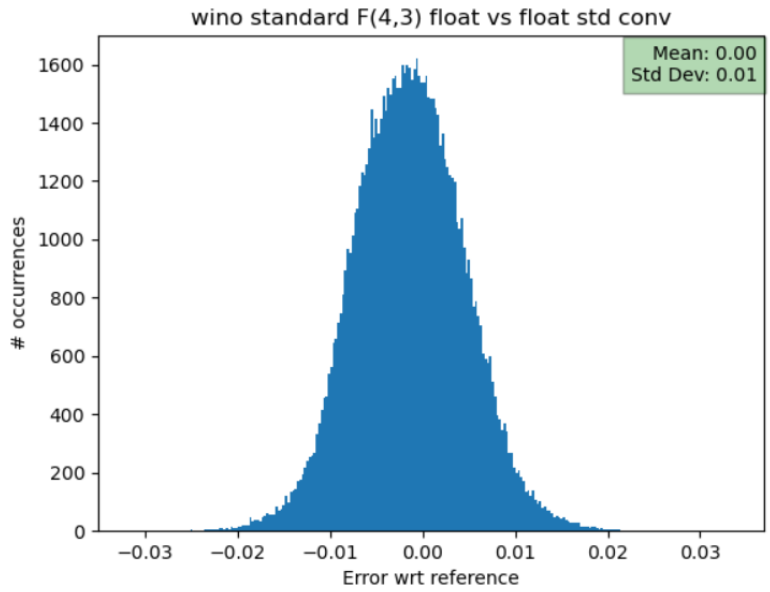


Figure 3.2: Distribution of the error of the Winograd convolution result with respect to the standard convolution, with data stored in floating point format.

3.1.3 Quantized Winograd F(4,3) algorithm

The Winograd algorithm can also be used to compute convolutions when the data is stored in integer format, which is the standard for quantized data, with the only additional consideration that the transformation matrices must be scaled for integer arithmetic [22]. For the case of the `F(4,3)` algorithm, the activations and output transformation matrices `B` and `A` are already composed by integer values, so the only matrix that must be scaled is the weight transform matrix `G`. The scaling factor is derived from the largest denominator, that is in this case 24, and applied element-wise to all elements. The new transformation matrix `G'` is

$$G' = 24 * G = \begin{bmatrix} 6 & 0 & 0 \\ -4 & -4 & -4 \\ -4 & 4 & -4 \\ 1 & 2 & 4 \\ 1 & -2 & 4 \\ 0 & 0 & 24 \end{bmatrix} \tag{3.7}$$

Some considerations must be made about operations with data stored in integer format.

A problem specific of integer data and that must always be taken into consideration is that to correctly store the result of an addition or multiplication operation the output bitwidth must be larger than those of the two inputs, otherwise the result risks not to be stored properly and to incur in overflow problems. Specifically, the addition between two values stored on ζ bits requires up to $(\zeta+1)$ bits, while the multiplication between two values on γ and δ bits respectively requires $(\gamma+\delta)$ bits. Since the transformation operation, as well as the element-wise matrix multiplication, increase the bitwidth of the data, the accelerator internal parallelism will progressively increase as the Winograd convolution is computed. Moreover, its output activations will be stored on a larger number of bits than the input activation, and their bitwidth must be must be reduced to the accelerator nominal one at the end of the computation. This procedure induces an additional quantization that can introduce a numerical error with respect to the standard convolution result and therefore induce a network accuracy reduction.

Considering the largest value of the filter transformation matrix reported in (3.7), 24, it is possible to derive that the transformed filter elements require a bitwidth increase of $\lceil log_2(24^2) \rceil = 10$ bits. A similar reasoning can be done for the activations, that require a bitwidth increase of $\lceil log_2(5^2) \rceil = 5$ bits. Moreover, as discussed in section 3.1.2, the presence of numbers that are not power of 2's has an important impact on the transformation arithmetic complexity.

Finally, the analysis of the behaviour of this version of the algorithm coupled with integer data stored on 8 bits shows that this solution is not suited for applications that require a high accuracy, since a significant error is introduced in the computation. Table 3.1 reports that the Python model evaluates for the reference layer an output error with a mean of 7.23 and with a standard deviation of 28.14, whose distribution can be seen in figure 3.3.

3.1.4 Data transformation

Previous sections have highlighted how the use of the Winograd algorithm comes with the requirement of transposing the data into the Winograd domain through a transformation. While the input activations must be transformed in real-time because they are not known before the start of the computation, two different choices can be made for the weights:

- real-time weight transformation
- off-line weight transformation

Off-line weight transformation

Differently from what happens for the activations, once the training of the network is complete the weight values are known and fixed. When the allocation of resources for the weights transformation is a problem, the design might contemplate the possibility to

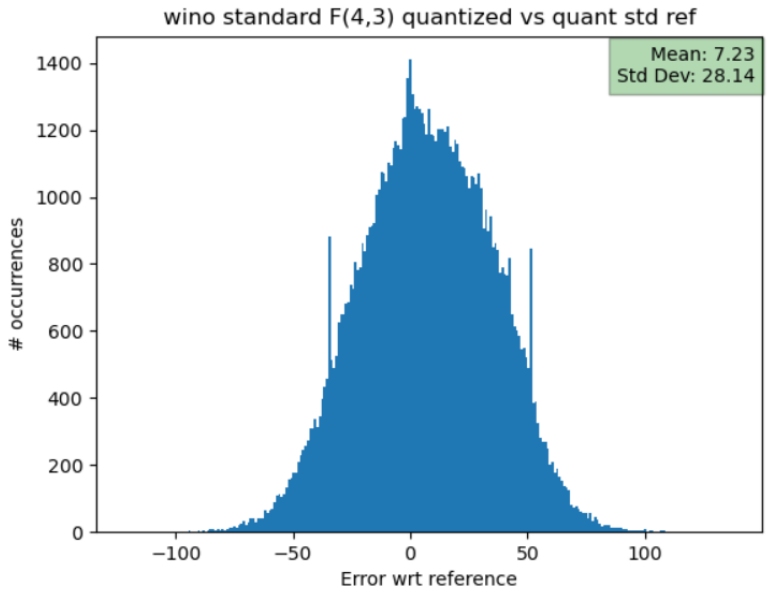


Figure 3.3: Error distribution of the `F(4,3)` quantized Winograd algorithm, used with integer format, 8-bit quantized data, with respect to the result of the standard convolution performed on the same data.

transform the weights before the start of the computation, so that the values loaded from the memory can be directly used during the convolution operation without the need for additional manipulation. Offline transformation also helps to reduce the quantization error because the quantized weight values can be computed by performing the transformation with 32-bit floating point values and by then quantizing an accurate result. While this solution leads to the use of fewer computational resources, it also significantly increases the memory requirements. Indeed the transformation process induces an increase of both

- the number of values that need to be stored, because each (r x r) filter becomes, in the Winograd domain, a (n x n) filter;
- the data bitwidth, as discussed in section 3.1.2.

One must also take into account that the memory increase will concern both the off-chip memory that store all the layer parameters, and therefore increasing the cost of the memory access both in terms of time and energy, and the on-chip memory on which the data must be loaded during operation. In particular, for the case of the `F(4,3)` algorithm, the amount of additional required storage space can be computed as $10\ bits \cdot N_{kx} \cdot N_{ky} \cdot N_{if} \cdot N_{of}$.

Real-time weight transformation

Performing the weight transformation during the operation requires the allocation of computational resources specifically dedicated to this task. This can however be an effective choice because as discussed above it helps in reducing the off-chip memory requirements.

The analysis does not take into account the time required to perform the transformation, because in the case of real-time weight transformation this computation would in any case be done in parallel to the image tile transformation and would therefore not have an impact on the accelerator throughput. Moreover, if the three steps of the Winograd convolutions are connected as a pipelined structure the three transformations do not increase inference time.

Image tile transformation

As was stated before, there is no possibility for off-line transformation for the layer activations. Some considerations must be nevertheless made also in this regard.

When the current layer requires the computation of multiple output channels, each image tile must be reused multiple times, as already happens for the weights: therefore also in this case two different solutions can be exploited during the design. The first possibility is that of applying the transformation to the image tile every time it is fetched; the second solution is that of transforming each image tile only once and then store the transformed values into an on-chip buffer until it is not needed anymore for the computation. Hybrid solutions where the transformed tiles are stored and reused for a limited number of iterations before being discarded are also possible.

This design will not store the transformed tiles, thus applying the transformation as many times as needed, but will store the transformed weights.

3.2 Considerations on convolution acceleration on FPGAs

Two main difficulties can be highlighted with respect to the design of a CNN accelerator on an FPGA. The main difficulty is avoiding a computing throughput degradation due to the under-utilization of either the logic or the memory bandwidth that are available on the FPGA platform [24]. Different optimization techniques, such as loop pipelining, loop tiling and loop unrolling are often exploited to improve the accelerator throughput through the maximization of the resources utilization. Secondly, FPGAs do not offer an on-chip memory space large enough to store the large amount of parameters that are used to describe CNN layers as well as the initial and intermediate activations: FPGA-based accelerators must therefore rely on an off-chip DRAM.

3.2.1 Memory hierarchy

Communication with an external memory, with respect to on-chip memories, is always characterized by a longer access time and a higher energy consumption associated to the movement of data. In addition to this, because of how the convolution operation is defined, in the specific case of a CNN accelerator both activations and weights must be reused multiple times, and therefore must be accessed multiple times.

To achieve at the same time an increase in throughput and a reduction of the energy consumption, FPGA-based accelerators must limit the number of memory accesses: to achieve this, they usually implement three main memory levels:

- External memory
- On-chip buffers
- Registers

During operation, the data is loaded from the external DRAM into the smaller on-chip buffers, from which it can be more readily retrieved. The data on the on-chip buffers must discarded only once it is not needed anymore so to avoid fetching the same activations multiple times.

External memory organization

To minimize and standardize the number of memory accesses, and subsequently enhance the accelerator efficiency, the data inside the DRAM memory should be organized with a fixed pattern, independent on the layer parameters. Data streams should also have a fixed bandwidth.

3.2.2 Loop Tiling

Tiling is an efficient way of loading a spatial sub-set of data into the on-chip buffers. The idea consists into dividing the input data into blocks that can be separately loaded into the local memory space and that the accelerator can interpret as stand-alone ifmaps on which perform the convolution. When coupled with the use of ping-pong buffers in a pipelined structure tiling can also be used to mask the external memory access time during the operation, so that while the first tile is used a second one can be loaded into the second buffer.

Loop tiling requires the introduction of new design variables:

- T_{ix} and T_{iy}: spatial dimensions of the tile generated from the ifmap, $T_{ix} \leq N_{ix}$, $T_{iy} \leq N_{iy}$
- T_{if}: number of channels, $T_{if} \leq N_{if}$

- T_{ox} and T_{oy}: spatial dimensions of the output tile generated by the convolution between the layer filter and the tiled ifmap, $T_{ox} \leq N_{ox}$, $T_{oy} \leq N_{oy}$
- T_{of}: number of output channels, $T_{of} \leq N_{of}$

Remembering equation 2.4, and assuming N_{ix} and N_{iy} to already include the zero padding, these variable are related by the relation [10]

$$\begin{aligned} T_{ix} &= (T_{ox} - 1) \cdot s + N_{kx} \\ T_{iy} &= (T_{oy} - 1) \cdot s + N_{ky} \end{aligned} \tag{3.8}$$

Once the design tiling factors are set, the design can allocate for the ifmap tile a buffer of size $T_{ix} \times T_{iy} \times T_{if} \times datawidth$. Assuming $T_{if} = N_{if}$, the number of tiles Θ required to process the whole ifmap can be computed as:

$$\Theta = \left(1 + \left\lceil \frac{N_{ix} + 2 \cdot p - T_{ix}}{T_{ix} - r + 1} \right\rceil\right)^2 \tag{3.9}$$

3.2.3 Unrolling

Loop unrolling is a technique that allows to parallelize the computation, thus increasing the throughput at the cost of increasing the amount of allocated hardware resources.

Equation 2.6 reports the number of multiply operations required to compute the result of the convolution operation. When each multiplication is processed separately, this is also the number of required iterations. Algorithm 1 clearly shows how the convolution operation can be divided into four main loops:

1. loop over the number of weights of every kernel, $(N_{kx} \times N_{kx})$;
2. loop over the number of input channel N_{if};
3. loop over the number of output activation of every ofmap, $(N_{ox} \times N_{oy})$;
4. loop over the number of output channels N_{of}.

The analysis of each of these loops leads to different loop unrolling opportunities as well as different possible data reuse patterns [10]. In the following we will make use of the parameters P_{if} and P_{of}, that represent the number of parallel computation along the number of input and output channels respectively. As already illustrated for the tiling factors, these unrolling factors must be compliant with the relation

$$\begin{aligned} 1 \leq P_{if} \leq N_{if} \\ 1 \leq P_{of} \leq N_{of}. \end{aligned} \tag{3.10}$$

Inner loop unrolling

The inner loop of the convolution is the one for which the computation can be more easily parallelized, because it requires the computation of $(r \times r)$ multiplications, that can be easily performed in parallel since they do not present any data dependency.

Assuming to have a $(r \times r)$ kernel, unrolling the inner loop would require the parallel computation of $r \cdot r$ multiplications. The computation of each output value would therefore require N_{if} iterations instead of $Nif \cdot N_{kx} \cdot N_{ky}$. The computing architecture presents $r \cdot r$ multipliers working in parallel and an adder three for the results accumulation.

Unrolling over P_{if} channels

The number of iterations can be further reduced by processing P_{if} sets of activations and weights, taken from the same (x, y) position from different ifmaps, in parallel. The computing architecture would not be modified with respect to the previous case, because all the computed products result must be accumulated in both cases.

Unrolling the loop over the input channels with factor P_{if} requires the execution of $P_{if} \cdot N_{kx} \cdot N_{kx}$ in parallel and reduces the number of iterations by a factor of P_{if}.

Outer loop unrolling

The loop over the number of output channels can be unrolled similarly to the one over the input channels. Unrolling this loop with factor P_{of} means performing the multiplication of one activation with P_{of} weights taken from the same (x, y) position of different kernels and allows to compute P_{of} partial outputs.

Coupling the unrolling of the outer loop with those of the inner and input channel loops requires a number of parallel MAC operations Ξ

$$\Xi = N_{kx} \cdot N_{ky} \cdot P_{if} \cdot P_{of} \tag{3.11}$$

which is also equal to the number of DSPs that need to be allocated to work in parallel.

Unrolling of Winograd convolution

The unrolling strategies described above can also be used to accelerate a Winograd convolution. In this case the number of parallel MAC operations will be equal to

$$\Xi = r \cdot r \cdot P_{if} \cdot P_{of} \tag{3.12}$$

Assuming to use the `F(4,3)` complex Winograd algorithm coupled with Karatsuba's algorithm and unrolling factors P_{if} and P_{of}, the computation of $(4 \times 4 \times P_{of})$ output values requires $46 \cdot P_{if} \cdot P_{of}$ parallel DSPs.

3.2.4 Loop pipelining

Loop pipelining is a common technique used to decrease the iteration latency of a computing system. In a pipelined structure the computation is split into a series of sequential steps that can be overlapped in time, or executed in parallel. This architecture is characterized by two main parameters:

- Iteration Interval (II): number of cycles that pass between the start of two contiguous iterations
- Iteration Latency: number of cycles required to complete each iteration

Most HLS tools are able to autonomously pipeline the execution of loop through the allocation of the required resources, among which the most important are the pipeline registers required to correctly time the execution. Moreover, they are also able to detect possible data dependencies and implement the required control logic.

3.3 Motivation

Real-time CNN inference is becoming of fundamental importance for many difference fields, and a lot of research is being done on the design and development of a fast and efficient hardware accelerator.

The hardware support of choice is often an FPGA, but the design of efficient accelerators presents many challenges in terms of both optimization of resources utilization and of acceptable throughput. Winograd minimal filtering algorithms have been largely exploited to accelerate convolutions with small kernel sizes, but the improvements that these algorithms are able to bring in terms of throughput come at the expense of additional logic, required to perform the transformation into and from the Winograd domain. Moreover, this approach lacks flexibility in terms of supported filter dimensions as well as non-stride-1 convolution. Finally, when coupled with data quantized on 8 bits, Winograd algorithms introduce a significant numerical error in the computation, and therefore strongly reduce the network prediction accuracy.

This thesis presents the description of an FPGA-based hardware accelerator for quantized data on 8-bits based on the complex `F(4,3)` Winograd algorithm. The use of this newly-derived complex version of the Winograd `F(4,3)` algorithm introduced in [22], coupled with Karatsuba's algorithm and a particular data organization [22], can help in effectively reducing the computation error while only slightly decreasing the gain in terms of number of multiplications provided by the Winograd convolution, also when the data is quantized on 8-bits. Flexibility in terms of kernel size can be achieved with minimal resources overhead thanks to a matrix reuse paradigm, while flexibility in terms of stride value can be obtained thanks to a specific decomposition method proposed by [8]. Loop unrolling real time transformation of data allows to reduce the memory demands. Tiling

and data reuse, as well as an efficient memory access paradigm, are exploited to improve the overall performance through the design of a specific component that loads the inputs on a buffer that presents three degrees of flexibility and is able to process different ifmap dimensions.

Chapter 4

Related Work

A lot of research has focused on the design of efficient and flexible CNN accelerators: this chapter reports the works that have acted as starting point for the development of the current thesis.

4.1 WRA: A Highly Unified Dynamically Reconfigurable Accelerator

This work [8], presented in 2019, describes the implementation of an FPGA-based CNN accelerator that is based on the standard `F(2,3)` Winograd algorithm and that supports a broad range of configurations in terms of filter size, stride value and types of operations.

The authors selected the `F(2,3)` algorithm because of its hardware-friendly transformation matrices and developed a decomposition method to improve flexibility in terms of supported kernel sizes and stride. Convolutions with kernel sizes with $r > 3$ are mapped into a convolution with a kernel with size (3×3) by decomposing the kernel into a variable number of sub-matrices. This allows to use the same hardware resources allocated for the `F(2,3)` also when $r \neq 3$. This decomposition method can however lead to multipliers in-utilization when the kernel dimension is not a multiple of 3. A similar decomposition method is exploited also to map non-stride-1 convolutions into stride-1 convolutions, that can therefore be accelerated with the same hardware resources. Finally, the convolution with kernels of size (1×1) and (2×2) is instead executed as standard convolutions and so require the allocation of a dedicated hardware support.

This accelerator presents many degrees of computation parallelization. Data reuse and high throughput are achieved thanks to a flexible buffer used to store the activations. Finally, the computation of the Winograd convolution is entrusted to an array of Processing Elements that exploit real-time transformation for both activations and weights.

4.2 WinoCNN: Kernel Sharing Winograd Systolic Array

The accelerator described in this paper [15] is able to support the convolution with different kernel sizes thanks to the identification of a group of Winograd algorithms characterized by nearly-identical transformation matrices. The resources allocated for the transformation can therefore be reused independently of the layer parameters. This work also highlights how this class of algorithms is characterized by the same tile dimensions in the transformed domain, making the computation of the Winograd convolution invariant with respect to the kernel size as well.

4.3 Efficient Winograd Convolution via Integer Arithmetic

This paper [22], also published in 2019, provides three important contributions that can be used to improve the overall behaviour of Winograd convolution if exploited for the design of an accelerator. First, it introduces a new version of the Winograd `F(4,3)` algorithm derived by extending the interpolation points field to the complex field $\mathbb{C}$. Secondly, it describes optimization techniques that allow to reduce the number of multiplications required by the complex algorithm. Finally, it analyses a hardware-friendly precision scaling scheme that can be used to reduce the bitwidth increase of the inputs due to the transformation when the data is stored in integer form.

4.4 Summary

Both these works represent an important stating point for the design of the current CNN accelerator, which combines the main novelties of the three contributions, namely stride-2 decomposition, and the newly-introduced complex Winograd algorithm.

Chapter 5

Methodology

This chapter illustrates the optimization techniques that have been used in the design of WinoAdapt CNN hardware accelerator.

5.1 Complex Winograd

Meng et. al [22] have derived a new version of the `F(4,3)` algorithm by extending the field from which the transformation matrices interpolation points can be chosen from the field of rational values $\mathbb{Q}$ to the field of complex numbers $\mathbb{C}$. The new interpolation points become $[0, 1, -1, i, -i]$ and give the transformation matrices

$$B^T = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & -1 & 1 & -1 & 1 & 0 \\ 0 & -i & -1 & i & 1 & 0 \\ 0 & i & -1 & -i & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 1/4 & 1/4 & 1/4 \\ 1/4 & -1/4 & 1/4 \\ 1/4 & i/4 & -1/4 \\ 1/4 & -i/4 & -1/4 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & i & -i & 0 \\ 0 & 1 & 1 & -1 & -1 & 0 \\ 0 & 1 & -1 & -i & i & 1 \end{bmatrix} \tag{5.3}$$

Model	Data format	Ops ratio	Maximum difference	Mean	Standard deviation
Complex Winograd	Floating point	2.91	0		
Complex Winograd	8-bit Quantized	2.92	14	0.36	2.98

Table 5.1: Analysis of the results of Complex `F(4,3)` Winograd algorithm with respect to the standard convolution results, for both floating point and quantized data. The table reports the maximum difference between the analyzed Winograd models and the standard convolution and, when relevant, the error mean and standard deviation. The test layer has dimensions $N_{ix} = 54$, $N_{iy} = 54$, $N_{if} = 32$, $N_{of} = 32$ and is convolved with a (3×3) filter and stride 1.

The same transformation matrices can be used also to manipulate data stored in integer format, with the same considerations already reported for the standard algorithm.

The complex `F(4,3)` algorithm presents two main advantages with respect to the previously discussed one. With respect to the ones reported for the standard algorithm in 3.4, 3.5 and 3.6, these new matrices require a lower arithmetic complexity: the transformation of the activations and outputs can be computed with additions and subtractions only, while the transformation of the weights requires multiplications and divisions with numbers that are power of 2's and that can be computed with simpler shifts. Finally, the largest denominator has now been lowered to 4, which means that to take into account the data bitwidth increase the system only requires additional $\lceil log_2(4^2) \rceil = 4$ bits instead of the 10 bits of the standard algorithm. A lower internal bitwidth of the data allows to reduce both the amount of resources needed to perform the computation and the energy required by the hardware support.

The use of the complex field allows to significantly reduce the error introduced during the computation. The results reported in table 5.1 clearly show how for floating point data the Winograd convolution returns the same exact result of the standard computation, while for 8-bit quantized data the error is significantly lower than the one introduced during the computation of the convolution with the standard algorithm and already described in table 3.1. Figure 5.1 reports the error distribution of quantized data.

The choice of the complex field is based on two properties of $\mathbb{C}$ that are the symmetry and information redundancy, characteristic of complex numbers.

5.1.1 Complexity analysis

The use of complex transformation matrices, alongside the advantages already highlighted, leads to the presence of complex values into the transformed input and weight tiles. This means that in the Winograd domain the data representation will require twice the bitwidth,

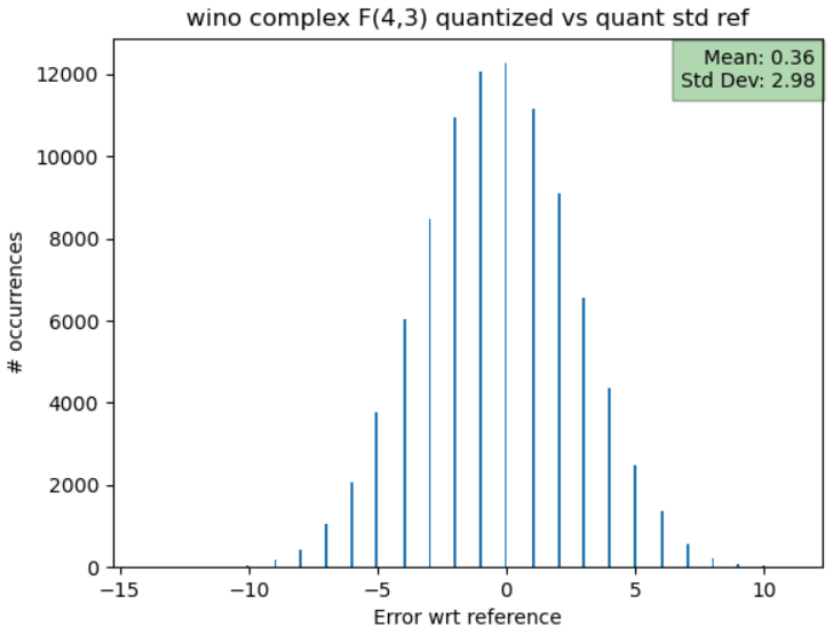


Figure 5.1: Distribution of the error introduced when computing the convolution with the complex F(4,3) algorithm and data stored in 8-bit integer format, with respect to the standard convolution with 8-bit quantized data.

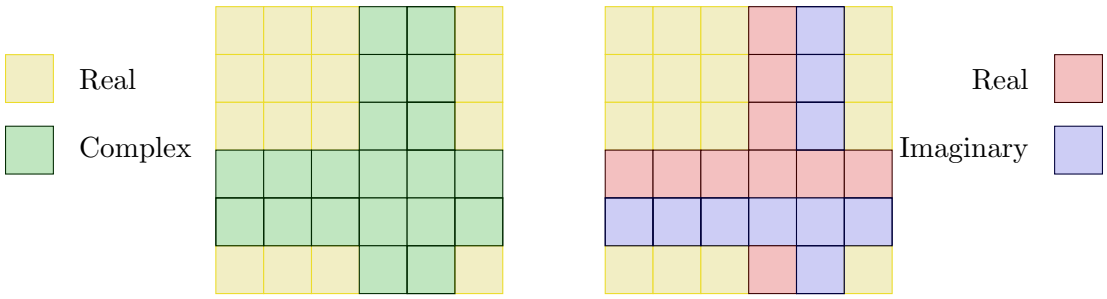


Figure 5.2: Graphical representation of the complex data organization. Left: Disposition of real and complex values in the transformed image tile and filter. Right: Example of a way to store complex conjugates values so not to have memory overhead.

actively doubling the required memory space. Moreover, multiplications with complex numbers require four multiplications instead of one and therefore require the allocation of a significantly larger amount of computing resources. Even though the complex algorithm could improve the accuracy degradation, additional optimizations of the algorithm are required before complex Winograd can effectively be used in the design of a CNN accelerator that satisfies the constrains in terms of resources and computation time.

5.1.2 Complex data organization

Meng *et al.* [22] have demonstrated that it is possible to store the complex transformed values without incurring in any memory overhead with respect to the standard Winograd case. Their analysis is based on the `F(4,3)` complex algorithm and starts by deriving that 16 elements out of the total 36 transformed image tile $D = B^T dB$ elements are real values; the other 20 present both a real and an imaginary part. The disposition patter of these values is fixed and can be found on the left in figure 5.2. Moreover, the analysis of the complex values shows how they can be organized as ten couples of complex-conjugates values so that the transformed tile becomes

$$D = \begin{bmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} & \overline{D_{0,3}} & D_{0,5} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} & \overline{D_{1,3}} & D_{1,5} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} & \overline{D_{2,3}} & D_{2,5} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} & D_{3,4} & D_{3,5} \\ \overline{D_{3,0}} & \overline{D_{3,1}} & \overline{D_{3,2}} & \overline{D_{3,3}} & \overline{D_{3,4}} & \overline{D_{3,5}} \\ D_{5,0} & D_{5,1} & D_{5,2} & D_{5,3} & \overline{D_{5,4}} & D_{5,5} \end{bmatrix} \tag{5.4}$$

The matrix contains only 36 independent elements and it is therefore possible to store the data using the configuration described on the right in figure 5.2. The same number of complex-conjugates pairs and data distribution pattern can be found in the transformed

filter $W = GgG^T$ and therefore the same data organization can be found also in the matrix $W \odot D$.

5.1.3 Karatsuba's algorithm

Additional considerations can be made about the number of multiplications. Considering the multiplication between two complex numbers $\alpha+j\beta$ and $\gamma+j\delta$ and the product between their complex conjugates $\alpha-j\beta$ and $\gamma-j\delta$, where α, β, γ and δ are real values, it is straight forward to derive that

$$\begin{aligned}(\alpha + j\beta)(\gamma + j\delta) &= (\alpha\gamma - \beta\delta) + j(\alpha\delta + \beta\gamma) \\ (\alpha - j\beta)(\gamma - j\delta) &= (\alpha\gamma - \beta\delta) - j(\alpha\delta + \beta\gamma)\end{aligned} \tag{5.5}$$

which means that the 20 complex multiplications can be halved exploiting the presence of the complex conjugates pairs and the EWMM can be computed with $16 + 4 \cdot 10 = 56$ multiplications.

The computation complexity can be further reduced if the complex multiplications are computed through the use of Karatsuba's algorithm. The idea is that of decomposing the multiplication parameters to underline a possible product reuse [22]:

$$\begin{aligned}(\alpha + j\beta)(\gamma + j\delta) &= (\alpha\gamma - \beta\delta) + j(\alpha\delta + \beta\gamma) = \\ &= (\alpha\gamma - \beta\delta) + j((\alpha + \beta)(\gamma + \delta) - \alpha\gamma - \beta\delta)\end{aligned} \tag{5.6}$$

This effectively reduces the number of required multiplications from four to three, meaning that with respect to the previous analysis the number of required multiplications to compute the EWMM for the `F(4,3)` complex algorithm becomes $16 + 3 * 10 = 46$. Globally, the algorithm complexity reduction with respect to the standard convolution becomes

$$\Phi = \frac{144}{46} = 3.13\times \tag{5.7}$$

5.1.4 Transformed data bitwidth

Even if the complex algorithm bitwidth increase due to the transformation is lower than in the case of the standard algorithm, it can still have an impact on the amount of resources required to manipulate the data. Meng *et al.* [22] also proposed a precision scaling scheme that can be used to reduce the transformed weights bitwidth from 13 to 9 bits only. The use of these reduced-precision weights introduce an average proportional error of 0.1%, and therefore do not have a significant impact on the network final accuracy.

Their analysis also evaluates that when input activations are stored as unsigned integers on 8 bits, the transformed tiles presents 11-bit values. Therefore to take into account signed values the transformed activations values will be stored on 12 bits.

5.2 Flexibility through transform matrices reuse

The main limit to flexibility inside a Winograd-based hardware accelerator with respect to supported filter sizes lays in the constant transformation matrices. In order to be able to process multiple kind of layers, an accelerator should be able to select, during operation, the correct version of the Winograd algorithm and therefore the correct transformation matrices. Because the transformation is realized though a matrix multiplication, the most straight forward solution, that also allows the freedom to consider different matrices, is a DSP-based implementation of the transformation, which would however require the allocation of a large number of resources. For the reference case of the F(4,3) algorithm, the transformation of each (6×6) input tile requires the matrix multiplication with the (6×6) transformation matrix B. Each of the 36 transformed elements requires 6 multiplications to be computed, which adds up to 216 multiplications: this number is clearly higher than the 46 multiplications that are required to compute the EWMM and the analysis proves why the Winograd algorithm becomes inefficient for large image tiles, as described in section 3.1.1. Moreover, this solution would increase the transformation computation time because these operations cannot be performed in parallel and the intermediate results of the first Matrix Multiplication is needed to start the computation of the second MM. In addition, the number of multiplications is strictly linked with the specific algorithm that is being used, so depending on the parameters of the current layer some of the resources allocated for the transformation might be unused. Finally, the same considerations also hold for the output transformation and the required resources increase even more when the weights are transformed in real time as well.

The number of required resources can be largely reduced if the accelerator is designed for a specific Winograd algorithm: in this context, the matrix multiplication becomes a constant multiplication that presents a specific pattern that can be replicated with a purely logic-based implementation. However, in order to allow the accelerator to support multiple kernel sizes, i.e. multiple Winograd algorithm derivations, it is necessary to allocate a proper amount of transformation logic for every desired kernel size, since each transformation will be characterized by entirely different transform matrices and therefore a different constant paradigm.

This solution is sub-optimal also because different algorithms have different tile dimensions in the Winograd domain, so additional resources overhead would be added also for what concerns the EWMM.

Matrix reuse paradigm

Despite it being seemingly counter-intuitive, this logic-based solution can also be used in the context of a flexible accelerator thanks to a matrix reuse paradigm described by Liu *et al.* in [21]. This work shows how Winograd algorithms characterized by a constant value $\omega = m + r - 1$ can be used to implement a flexible hardware accelerator that supports different

kernel sizes without the need to allocate a large amount of logic for the transformation.

The following analysis will take into consideration the `F(1,6)`, `F(2,5)`, `F(3,4)`, `F(4,3)`, `F(5,2)` and `F(6,1)` algorithms. All these versions of the algorithm are characterized by the same value n = m + r – 1 = 6. Figure 5.3 shows the transformation matrices of the six algorithms and it easy to verify that they share the following properties:

- all algorithms share the same image tile transformation matrix B^T
- the filter transformation matrices G when $r < 6$ are scaled so that the transformed filter dimensions are equal to those of the inputs and therefore present only $r$columns. The elements of these columns are equal to those of the `F(1,6)` algorithm minus the bottom right element that is equal to 1
- the output transformation matrices A^T are as well characterised by different dimensions and present $m \leq 6$ rows. For algorithms with $m < 6$ the matrix values are equal to those of the first m rows of the output transformation matrix of the `F(6,1)` algorithm, minus the bottom right element that is again equal to 1.

Also for the case of Complex Winograd algorithms, when ω is fixed the transformation matrices present the same properties described above, as can be evaluated from figure 5.4, that depicts the transformation matrices of the complex `F(1,6)`, `F(2,5)`, `F(3,4)`, `F(4,3)`, `F(5,2)` and `F(6,1)` algorithms. As has already been highlighted in section 5.1, for this second case the transformation requires only addition and shift operations, completely removing the need for DPS blocks in addition to those required for the EWMM.

Flexibility can therefore be achieved if it is realized by varying the r and m parameters while keeping n constant. In this context the zeros and one values of the last column (or row) can be used as identifiers for the specific kernel size that the accelerator is currently working on.

The highlighted property can be exploited to implement a logic-based transformation that is flexible enough to be used for algorithms which are able to accelerate the convolution with filters of different dimensions reusing the same resources. Moreover, if this matrix-reuse paradigm is implemented, the Winograd-domain matrices will always have the same dimensions (in this specific case (6 x 6)), so the EWMM will be invariant of the kernel size and will be able to reuse the same architecture and always exploit all the allocated DSPs.

Theoretical MAC gain

The theoretical gain in terms of MAC operation reduction is not equal for all the considered algorithms. In particular, as depicted in figure 5.5, it significantly lowers for kernels with size $r = 1$ and $r = 6$, where the complex algorithm is shown to perform more multiplications than the standard convolution. However this slight performance degradation is still acceptable since for $r = 3$, which as already highlighted is the most common kernel

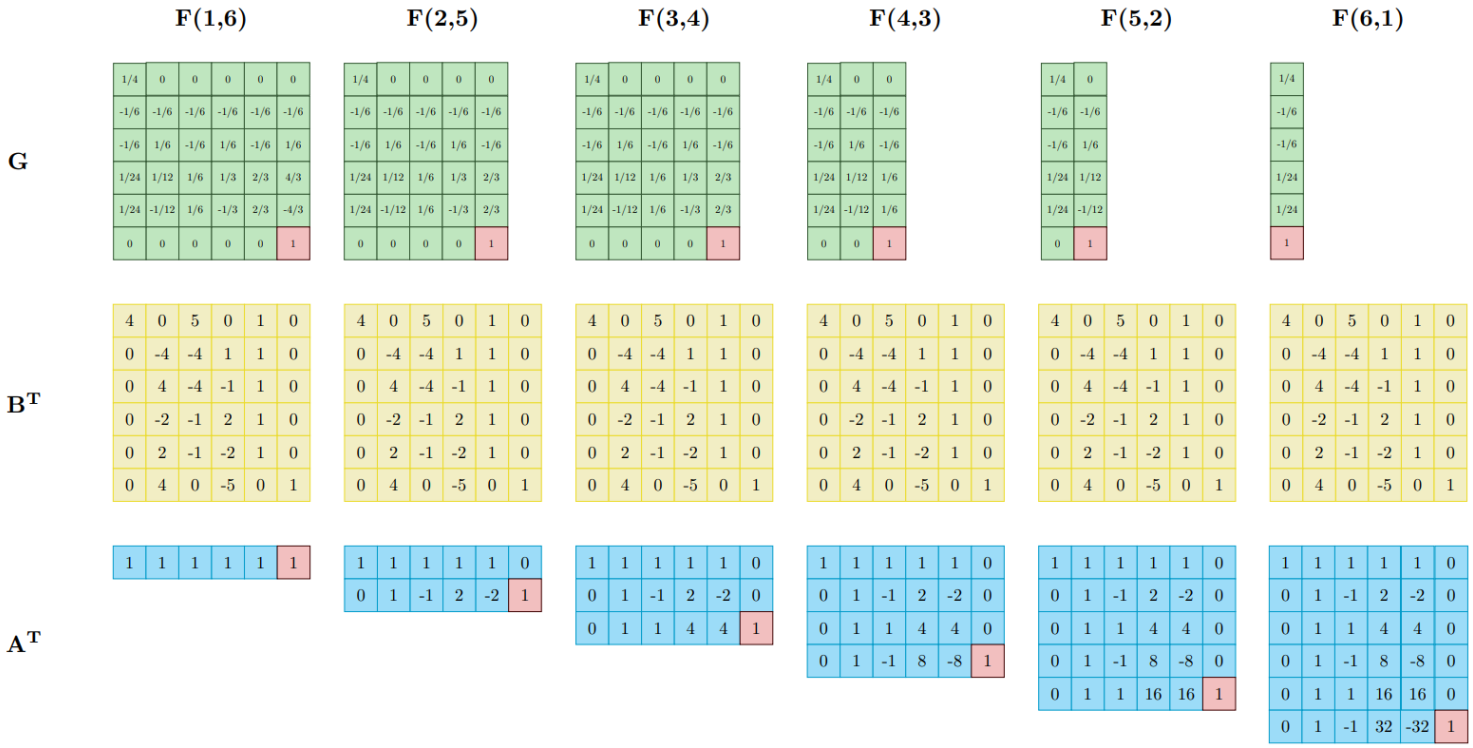


Figure 5.3: Transformation matrices of the standard `F(1,6)`,`F(2,5)`,`F(3,4)`,`F(4,3)`,`F(5,2)` and `F(6,1)` Winograd algorithms.

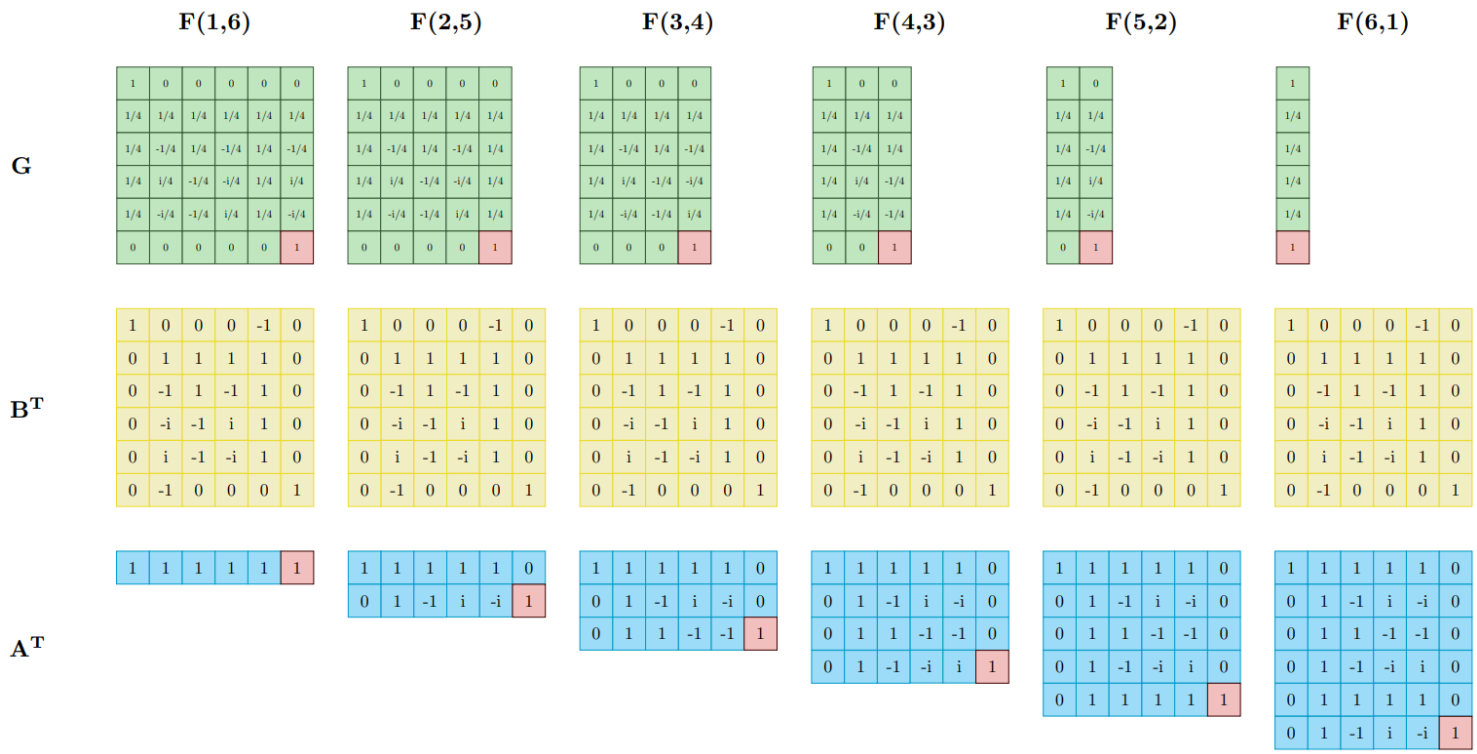


Figure 5.4: Transformation matrices of the complex `F(1,6)`,`F(2,5)`,`F(3,4)`,`F(4,3)`,`F(5,2)` and `F(6,1)` Winograd algorithms.

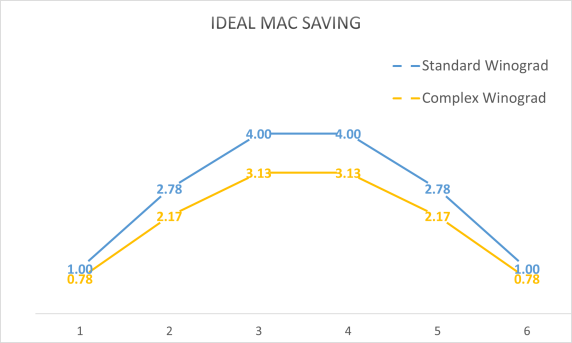


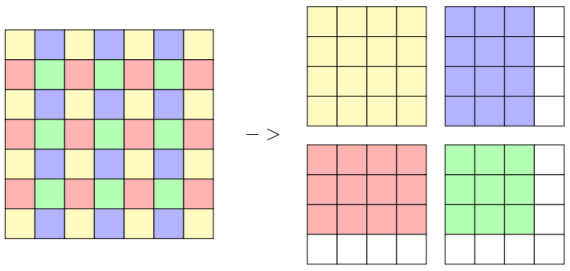
Figure 5.5: Theoretical MAC gain of the considered Winograd algorithms, as a function of the kernel size r.

dimension in the ResNet18 model, the gain with respect to the standard convolution is still significant.

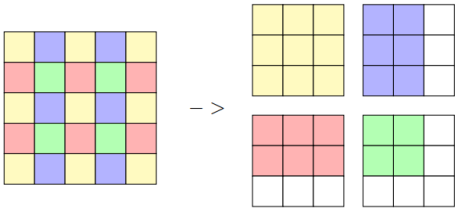
5.3 Stride-2 decomposition

The Winograd algorithm does not present a specific derivation to support the convolution with stride values different than 1. Yang *et al.* [8] have however proposed a Convolution Decomposition scheme, the Convolution Decomposition Winograd (CDW) method, that allows to process stride-2 layers with the use of the Winograd algorithm and the same hardware support used for stride-1 layers. CDW is based on the idea that when stride 2 is applied, even(odd)-index elements of the ifmap are always multiplied with eve(odd)-index elements of the kernel. The method can therefore be used to split the image tile into four new tiles, assigning each element of the original tile to one of the new tiles considering the even-odd nature of its indexes. The filter is as well split into four new filters, following the same procedure. The size of the decomposed tiles and filters can be arbitrarily chosen to refer the new convolution to a specific `F(m,r)` Winograd algorithm: any cell of the decomposed matrices that cannot be filled with an original matrix value are filled with 0's. A graphic description of the decomposition process is shown in figure 5.6, where a (7×7) tile is decomposed into four (4×4) tiles and a (5×5) kernel is decomposed into four (3×3) kernels; the white cells are empty and represent the padding required to have all the tiles of the same size. Once the four tiles and filters have been generated, the Winograd algorithm can be applied to each couple of same-color tile and filter separately. The final result is given by the accumulation of the four convolutions results. This decomposition method is able to transform a stride-2 convolution into a stride-1 convolution with equivalent parameters

$$r' \geq \left\lfloor \frac{r}{s} \right\rfloor \tag{5.8}$$



(a) Image tile decomposition.



(b) Filter decomposition.

Figure 5.6: Example of decomposition method with a (7 x 7) image tile and (5 x 5) filter.

$$n' = r' + m - 1 \tag{5.9}$$

$$n = 2 * n' \tag{5.10}$$

where r is the size of the filter of the stride-2 convolution, r' is the minimum size of the equivalent stride-1 convolution, n' is the size of the decomposed image tile and n is the size of the original image tile.

The four tiles can be seen as four different channels of an ifmap with dimensions ($N_{ix}/2$ x $N_{iy}/2$), and therefore it is possible to design an accelerator whose computational unit is transparent to the difference between stride 1 and 2 layers, removing the need for the allocation of resources specifically for stride-2 computation.

5.4 DSP use optimization

Typically, DSPs available on Xilinx FPGA platforms are optimized for processing a multiplication between data stored respectively on 18 and 27 bits, and then a successive accumulation on up to 48 bits [25]. However, when these DSPs are used to perform the MAC operation on values stored respectively on 9 and 12 bits, which is the case for Winograd weights and activations in the case of the `F(4,3)` complex algorithm, a significant portion of their logic remains unused during computation. Fu *et al.* [25] illustrated a way to optimize the DSPs utilization by using it to perform two concurrent multiplications, i.e. the multiplication of two values a and b with a same multiplicand c, when all the input data is

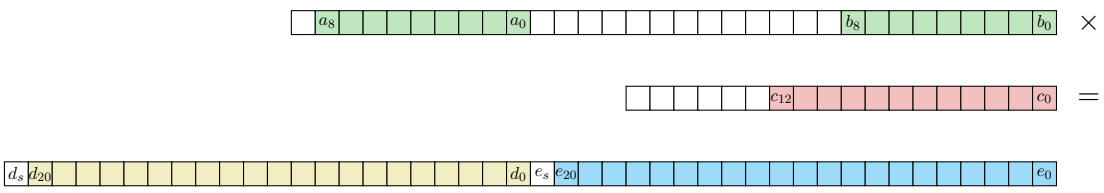


Figure 5.7: Graphical representation of the paradigm to map two multiply operations on one DSP.

stored on 8 bits. Their implementation exploits the DSP pre-adder to encode the a and b values in the 27-bit multiplicand in such a way that the output product can be computed exploiting the property

$$(a + b) \cdot c = a \cdot c + b \cdot c \tag{5.11}$$

The encoding paradigm must ensure that the multiplication outputs are fully distinguishable one from the other.

It is possible to use the described solution to design an encoding paradigm that allows to perform the multiplication of a 12-bit activation value c with two 9-bit weights a and b. The multiplication between a 9-bit value and a 12-bit value produces an output on 21 bits; an additional bit is required to store the result sign. This means that the encoding must ensure a 22-bit separation between the LSBs of the two weight values in the combined operand $(a+b)$. The first operand of the multiplication therefore requires $1+9+1+13+9 = 32$ bits. The second operand can be stored as an 18-bit value. However, to prevent an overlapping of the result bit values the actual bitwidth of the data stored on this second operand cannot require more than 12 bits. A graphical representation of the data organization inside the three vectors can be seen in figure 5.7, where

$$d = a \cdot c$$
$$e = b \cdot c$$

In order to ensure the correctness of the multiplication results, it is also necessary to implement a correction mechanism that adds the value 1 to the multiplication result saved in the output vector MSBs (in this case, d) when the input values b and c have opposite sign.

Chapter 6

Design principles

Previous chapters have highlighted both the importance of CNNs hardware accelerators, their requirements in terms of throughput and the difficulties of their design. This work presents the design of WinoAdapt, which is an FPGA-based hardware accelerator for CNNs that works with data quantized on 8 bits. The design is based on the `F(4,3)` complex Winograd algorithm and presents a good flexibility in terms of supported kernel sizes and stride. Alongside (3×3) kernels, WinoAdapt is also able to support the stride-1 convolution with kernels of size (1×1), (2×2) and (4×4). Stride-2 convolutions with filter sizes from (2×2) to (8×8) is supported as well with the additional requirement of a minor software manipulation of the input data. Contrarily to other state-of-the-art accelerators (such as [8]), WinoAdapt does not support standard convolution: this allows to dedicate all the available resources to the Winograd-accelerated convolution. Moreover, the flexibility in terms of supported layers is achieved with minimal resources overhead, because all the supported convolutions use the same hardware support. Tiling, loop unrolling and pipelining are also used alongside Winograd convolution to improve the overall performance.

6.1 Design flow

The device behaviour is described in C and the RTL is synthesized through Xilinx HLS tools, which make the design scalable for different Xilinx platforms.

Programming Xilinx Boards through HLS requires using a flow of different tools. Vitis HLS is the environment that allows to implement the C description of the design, test its behaviour through a simulation of the C description, and and translates the c-based hardware description into RTL Along with the RTL, the synthesis provides an initial estimation of the required resources in terms of logic, register, DSPs and memory. Vitis HLS finally also allows to simulate the RTL behaviour and compare this results with that of the C-simulation to ensure that the RTL behaves actually behaves as the HLL descrip-

Feature	ZCU104 board
Logic cells	504K
CLB flip-flops	461K
Max.distributed RAM	6.2 Mb
Total block RAM	11 Mb
DSP slices	1728

Table 6.1: Xilinx Evaluation Board ZCU104 features and resources [26].

tion. Synthesis constrains can be imposed with the used of special directives introduced by the string `#pragma HLS` and that can specify, for example, the desired array partition paradigm, potential loop unrolling, or the request for a pipelined or dataflow connection between loops or internal components.

Vitis HLS output RTL can then be coupled with Vivado to produce a binary file that can be flashed on the FPGA.

6.1.1 Board

The hardware support that has been chosen for this project is the ZCU104 board, that is one of Xilinx General purpose evaluation boards. developed for allowing rapid-prototyping. The board presents both FPGA-logic and a multiprocessor system-on-chip [26]. The FPGA-logic allows to synthesize user customized designs and therefore provides high flexibility and reconfigurability. Finally, the logic block is coupled with a high speed DSRAM memory [26], which is the usual choice for systems characterized by limited space. The board capabilities are enhanced by the quad core processing system (PS) and the dual-core real-time processor that characterize the SoC. The board hardware resources of interest for this project are illustrated in table 6.1.

6.2 Overview

The accelerator is composed of two separate elements, namely *host* and *device*. The device is the hardware component and is responsible for the convolution acceleration. The host is required to correctly program and drive the execution of the device through the computation of all the parameters needed to complete the convolution operation and that depend on the layer that is currently being processed. It is also responsible to drive the DMAs that feed weights and input activations to the device and receive the output activations.

Device structure

The device presents three main building blocks, called Tile Organization (TO), Weight Organization (WO) and Compute Unit (CU). TO and WO receive the input activations

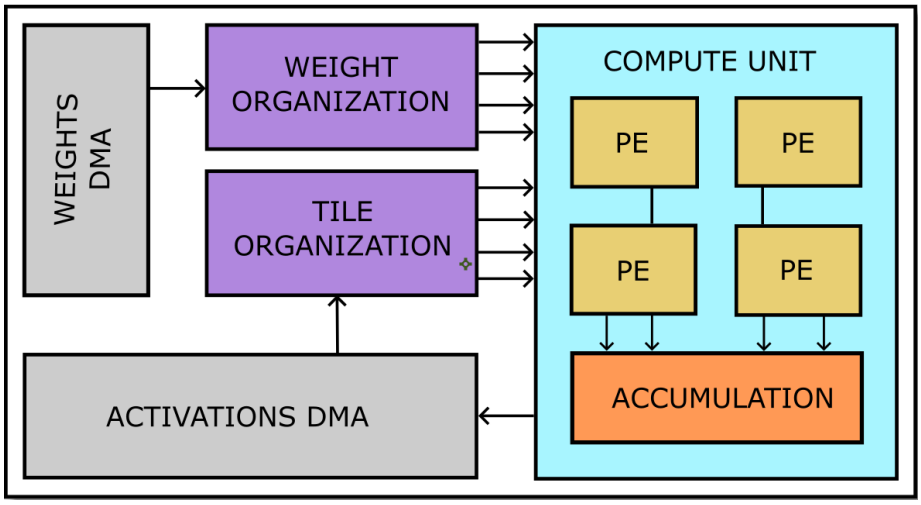


Figure 6.1: WinoAdapt device modules and internal connections.

and weights from two DMAs and are responsible for an initial manipulation of the data. Compute Unit is instead the component responsible for the convolution operation. The output activations are again collected by one DMA. The connection between the three components is shown in figure 6.1.

The main computational block of the design is a Processing Element (PE) that is able to perform all the steps required by the Winograd convolution and the accumulation of the output tiles along the input channel dimension.

6.3 Processing Element

The Processing Element is the core building block of the accelerator and represents the main computational unit of the design. It is a self-sustaining component that be used also on its own to compute stride-1 Winograd convolutions.

The PE is responsible for the transformation of input tiles and kernels into the Winograd domain; for the computation of the EWMM; for the output tiles accumulation along the input channel dimension; and for the inverse transformation of the output tiles. To make the design easier, each of these four tasks is performed by a different component. These elements are also connected through a task-level pipeline to increase the overall throughput. Finally, the four components are able to decrease latency thanks to loop unrolling, which is applied to all the most-inner computational loops with unrolling factors P_{if} and P_{of}.

6.3.1 Matrix reuse paradigm

The main characteristic of the PE is the ability to support the convolution with kernels of different sizes with only minimal resources overhead. This is achieved by allowing the design to implement different Winograd algorithms all characterized by the same $\omega = m + r - 1$ so to be able to exploit the transformation matrices reuse protocol described in section 5.2. The design is based on the `F(4,3)` complex algorithm, that sets

$$\omega = \omega_{F(4,3)} = m + r - 1 = 6. \tag{6.1}$$

This solution ensures that the image tiles, kernels and output tiles all have the same dimensions in the Winograd domain independently from the kernel dimensions, effectively making the element-wise multiplication invariant of the kernel size and removing any DSP overhead. The transformation matrices used for the design are the ones reported in figure 5.4.

6.3.2 Input Transformation

Input Transformation can transform P_{if} (6×6) input tiles into the Winograd domain at each iteration. This transformation is fully invariant of the kernel dimension because the input transformation matrices of algorithms characterized by the same ω are always equal. The transformation is implemented through a purely logical approach where the matrix multiplication manually described and as such fully unrolled. Both the input and output tiles always have dimensions $P_{if} \times n \times n$, with $n = \omega = 6$.

6.3.3 Weight Transformation

Weight Transformation transforms $P_{of} \times P_{if}$ kernels with size $(r \times r)$ at every iteration. After the transformation, these kernels have dimension (6×6) as the transformed input tiles. Also for this component, the transformation description is entirely logic-based. The matrix transformation is manually unrolled and four masks, defined by the current kernel size, are used to determine the correct size of the transformation matrix, which columns contribute to the transformation and the position of the bottom right 1 value.

To help make the design more flexible, the input data organization is independent of the kernel size and it is always interpreted as a group of $P_{if} \cdot P_{of}$ (4×4) kernels. For kernels with dimensions lower than (4×4), the weights are padded accordingly to figure 6.2.

6.3.4 Element Wise Matrix Multiplication

The Element Wise (EW) component performs the element-wise matrix multiplication between the transformed input tiles and weights. The multiplication is manually unrolled so to take into account the complex data organization described in section 5.1.1 and depicted in figure 5.4 and to force the system to implement Karatsuba's algorithm and exploit

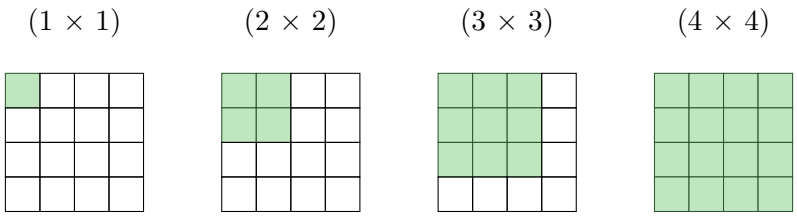


Figure 6.2: Graphical description of how padding must be applied to the weights depending on the kernel dimensions: the green cells represent the weight values while the white cells show the position of the padding 0's.

the presence of complex conjugates values: multiplications between real values uses one DSP, while multiplications between complex numbers requires 3 DSPs for every couple of elements. Finally, also this component makes use of the unrolling factors P_{if} and P_{of}. Recalling that each transformed image tile presents 16 real elements and 10 couples of complex-conjugates pairs, since only one multiplication will be mapped to each DSP the global DSPs requirement is expected to be

$$P_{of} \cdot P_{if} \cdot (10 + 3 * 10) = 46 \cdot P_{of} \cdot P_{if} \tag{6.2}$$

After the multiplication operation, the data is again stored with the same paradigm already illustrated for the transformed input and weights.

EW performs also the accumulation over N_{if} channels, so to reduce the number of inverse transformations that need to occur. The need to perform this accumulation requires however to take into account a possible bitwidth increase larger than the one expected in the case of the multiplication only: instead of $12 + 9 = 21$ bits, Element Wise results will be stored on 32 bits, which is the standard dimension of a DSP accumulator output.

Alongside the EW matrix multiplication, EW is also responsible for storing the transformed weight values and ensure weights reutilization. It has been chosen to store the transformed weights instead of their original 8-bit version because thanks to the complex algorithm the bitwidth increase due to the transformation accounts to only 1 bit. Each PE buffer is scaled to store up to $T_{ch} = T_{if} \cdot T_{of} = 1024$ kernels of dimension (4×4) with elements on 9 bits. The memory demand is therefore

$$4 \cdot 4 \cdot T_{ch} = 295\ kByte \tag{6.3}$$

At each iteration, the weights are either read from the stream, used for the computation and saved in the buffer, or read from the buffer to be used in the computation.

The dimension of this buffer poses a limitation on the number of input channels that the accelerator can process without the need for the host to perform additional software computations: the maximum N_{if} that the element can accept is equal to 1024 for stride-1 convolution and 256 for stride-2 convolution.

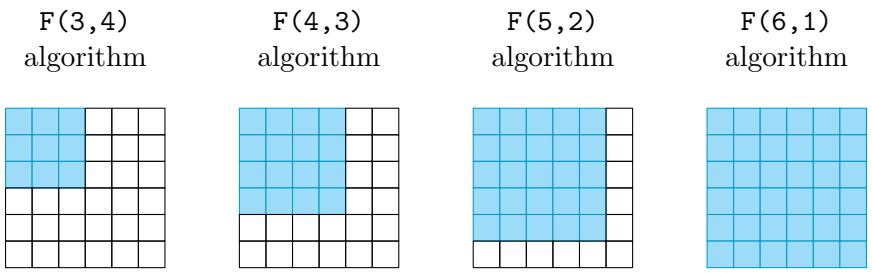


Figure 6.3: Graphical description of how padding is applied to the output image tiles: blue cells represent the activation values position while the white cells show the position of the padding 0's.

6.3.5 Output Transformation

Output Transformation implements the transformation in a similar way to Weight Transformation: the matrix multiplication is also in this case completely unrolled thanks to its manual definition and the component exploits a purely logic-based approach for the implementation. Again, the correct matrix dimension is selected through the definition of four masks that depend on the current Winograd algorithm and are able to mask the contribution of the unused rows. After the inverse transformation, the data bitwidth is increased to 37 bits to prevent overflow. At the same time, the size of the P_{of} tiles is reduced from (6×6) to $(m \times m)$, however the activations are padded so to always be organized as (6×6) tiles, i.e. the largest possible output tile that the processing element can produce. A graphical explanation of how the padding is implemented is reported in figure 6.3.

6.3.6 Internal Parallelism

As the previous sections have highlighted, the different components that make up the Processing Element all work with different bit-widths and in particular the data length increases as the computation proceeds. Initially, the activation and filter values are stored in memory as 8-bit integers and are fed to PE with the same format. The Input Transformation kernel receives activation values on 8-bits, but the data bitwidth is increased to 12 bits due to the transformation into the Winograd domain. For the same reasons, the weights bitwidth is increased from 8 to 9 bits inside the Weight Transformation kernel. The result of the element-wise matrix multiplication should be stored on 21 bits, but EW stores data on 32 bits to take into account the bitwidth increase due to the multiple multiplications and accumulations. Finally, the Output Transformation kernel further extends the bitwidth to 37 bits, due again to the transformation from the Winograd domain. However, the output data must be stored in the same format as the input data: the accelerator therefore requires a component responsible for reducing the data bit-width to the desired

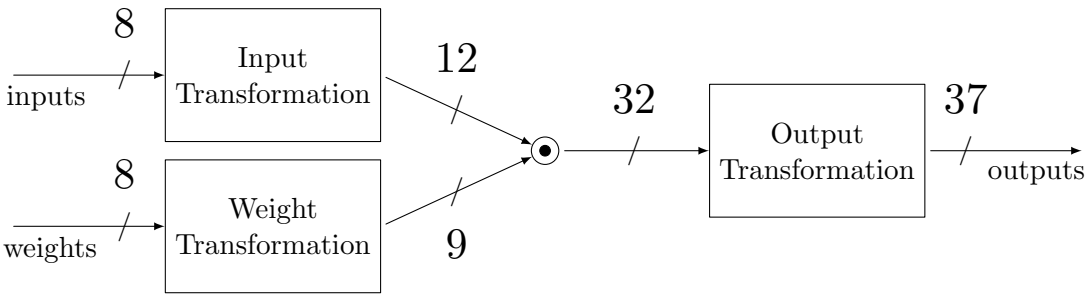


Figure 6.4: Processing Element internal parallelism.

value.

6.3.7 Hardware implementation

Transmission of data between different elements of the design can be described through the use of streams, a C++ template class provided by Vitis HLS. When the RTL is generates, streams are by default implemented as FIFO memories [27]. Streams are also used to define the interface of the device: while internal streams can have arbitrary dimensions, interface streams must be made of a number of bits equal to a power of 2's.

A PE connects to the nearby elements by means of three streams:

1. an input stream on $36 \cdot P_{if} \cdot 8$ bits that contains P_{if} input tiles of (6 x 6) elements on 8 bits
2. an input stream on $16 \cdot P_{if} \cdot P_{of} \cdot 8$ bits that contains $P_{if} \cdot P_{of}$ sets of (4 x 4) weights with values on 8 bits
3. an output stream on $36 \cdot P_{of} \cdot 37$ bits that contains P_{of} (6 x 6) tiles with elements on 37 bits.

6.4 Compute Unit

Compute Unit (CU) is composed of four Processing Elements that work in parallel and an Accumulation component that receives and accumulates the four PEs outputs to compute the final result. This particular architecture has been chosen because it helps in fulfilling both the desire to accelerate stride-1 convolution and the aspiration to support stride-2 convolution, without memory or resources overhead.

Stride-1 convolution

In the case of stride-1 convolutions, each PE process $N_{if}/4$ image tiles, referring to the same ifmap (x,y) positions but at different input channels; because the final output is given by the sum of the convolution of all channels, the four results must at the end be accumulated. Indeed, this four-PEs structure allows to process the four sets of $N_{if}/4$ input channels at a time, effectively speeding up the four computations, that are completely independent, with respect to the case where only one PE is allocated.

Globally, in the case of stride-1 convolution at each iteration Compute Unit processes $4 \cdot P_{if}$ (6×6) image tiles characterized by the same (x,y) position but different channels.

Stride-2 convolution

This peculiar organization is also efficient for computing stride-2 convolution with the decomposition paradigm described in section 5.3, that requires the computation of four separate convolutions and the accumulation of the results. In this case, each PE is responsible for the equivalent stride-1 convolution of one of the four decomposed input feature maps; therefore they each process one image tile of for all P_{if} input channels.

Therefore, in the case of stride-2 convolution Compute unit processes, at each iteration, P_{if} image tiles with dimension (12×12).

The behaviour of the Compute Unit is independent on the layer stride s and requires to know only the parameters r_{eq} and m, where

$$r_{eq} = \begin{cases} r & \text{if } s = 1 \\ \lceil \frac{r}{2} \rceil & \text{if } s = 2 \end{cases} \tag{6.4}$$

6.4.1 Accumulation

The Accumulation component is the simpler element of the accelerator and serves two different purposes. As the name suggests, this element takes as inputs the four streams that are output of the four PEs and accumulates them.

This component is also responsible for scaling the results and thus reducing the data bitwidth: indeed, it receives numbers stored on 37 bits and produces an output on 8 bits. Currently, the scaling paradigm is implemented through a variable bit-shift that depends on a *scaling factor*. The value of the scaling factor differs between the different kind of layers that can be processed and is communicated to the component by the host along all the other computation parameters.

6.4.2 Hardware implementation

CU presents a large number of connections with the external world, because it receives two input streams for every PE, one containing the input activations and one containing

the weights. In addition to those, this presents an output stream on $36 \cdot P_{of} \cdot 8$ bits that contains P_{of} tiles with dimension (6 x 6) and elements on 8 bits.

6.5 Input Tile Organization

Tile Organization (TO) is the name of the element that receives the input activations from the external memory and is responsible for organising them so to correctly feed them to the four PEs.

This design must present a large degree of flexibility in terms of shape of the tile that can be loaded while also receiving data with a pattern that allows to define an efficient memory access paradigm that is, again, as much as possible independent from the current ifmap shape.

6.5.1 TO operation

To make the memory access paradigm flexible and efficient, Tile Organization receives the inputs in streams with a fixed dimension of 1024 bits, or streams of 128 values stored on 8 bits. This data is sequentially loaded from the memory considering the ifmap spatial dimensions taking all channel values for every pixel before moving to the next ifmap position. This makes the stream organization independent of the ifmap dimensions and the memory access efficient because the values con be loaded sequentially, without the need to jump from one address space to the other.

Inside the kernel, the data is organized into four on-chip buffers that can each store up to 14.400 8-bit values, or a global tile with dimensions ($30 \times 30 \times 64$). This organization is driven by the parameters T_{ix}, T_{iy} and T_{if}, that identify the dimensions of the current tile and where $T_{if} = N_{if}/4$ represents the number of ifmap channels that are store in each buffer; moreover, the organization is flexible in terms of tile shape because the buffers are fully scalable with respect to all number of channels T_{if}, values along the x-dimension T_{ix} and values along the y-dimension T_{ix}; the only limitations are that the four buffers must all have the same shape and that the number of ifmap channels must be a multiple of 4 $PEs \cdot P_{if} = 8$, so that each buffer stores a multiple of P_{if} channels - if the ifmap does not satisfy this second condition, it must be padded with an appropriate number of zeros. Figure 6.5 shows how the input data is organized into the four buffers: since the PE components assume an unrolling factor of $P_{if} = 2$, the data in divided in groups of two values. To reduce the complexity in terms of logic, the data is treated as $8\text{-}bit \cdot P_{if} = 16\text{-}bit$ values.

The element throughput is increased by implementing the on-chip buffers as four ping-pong buffers so to pipeline the loading data into the buffer and reading the image tiles operations. At the beginning of operation, TO starts loading the data into the first set of four buffers; once the current tile has been entirely loaded, TO starts generating the four streams, one from each buffer, that are connected to the four processing elements. At the

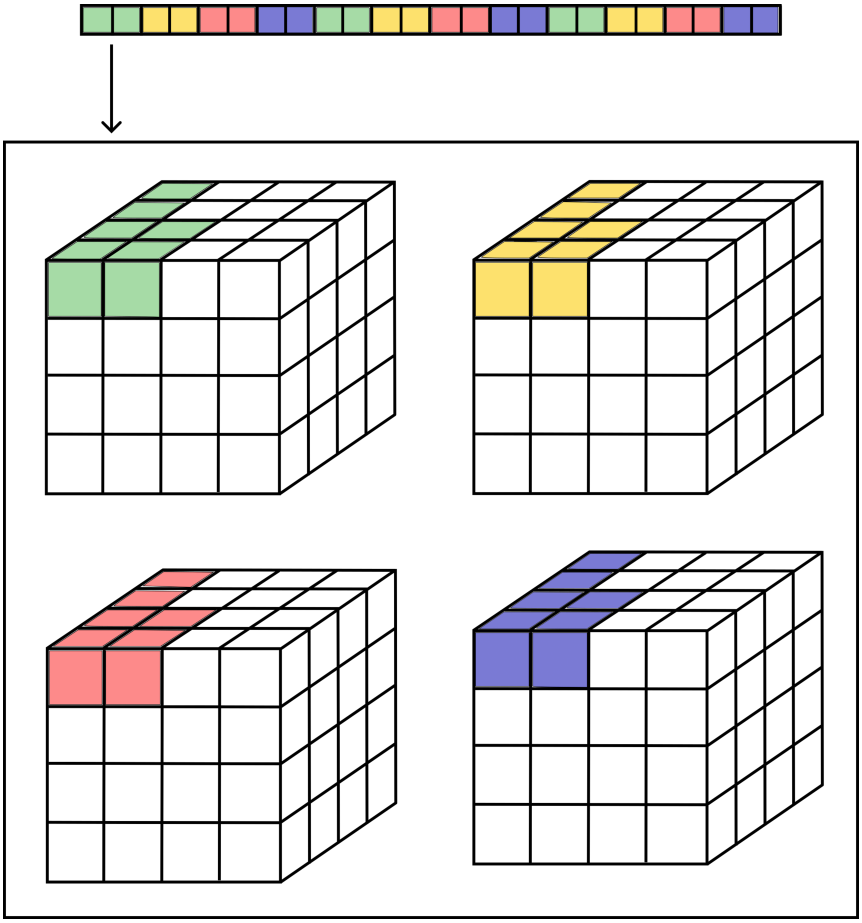


Figure 6.5: Graphical representation of how Tile Organization interprets the data read from the different input streams.

same time, the component starts loading the following tile into the second set of buffers, trying to mask the off-chip memory access time.

During the reading operation, the image tiles are generated by sliding a (6×6) window across the buffer with the equivalent Winograd stride $s_w = m$. The buffer read first in the channel dimension P_{if} elements at a time, then in the N_{ix} direction and finally in the N_{iy} direction, for N_{of}/P_{of} times before its content is discarded and a new loading operation begins.

6.5.2 Memory requirements

Tile Organization presents four sets of ping-pong buffers, where each buffer has dimension $M_{buff} = 14.400$ Byte, since they all can store up to 14.400 8-bit values. The element global

memory requirement is therefore

$$M_{TO} = 2 \times 4 \times M_{buff} = 115.200\ Byte = 921\ kb. \tag{6.5}$$

6.5.3 Hardware implementation

Tile Organization presents an input stream on 1024 bits and four output streams on 576 bits that contains P_{if} input tiles of (6 x 6) elements on 8 bits each. This components also takes as input all the variables needed to correctly drive its operation, which are:

- the tiling factors T_{ix}, T_{iy} and T_{if}
- the equivalent Winograd stride $s_w = m$
- the number of tiles Θ that need to be loaded to cover the whole ifmap, computed as described in eq. (3.9)
- the number of input streams that the system needs to read to fully load one tile, computed as $\rho = \lceil N_{if} \cdot T_{ix} \cdot T_{iy}/128 \rceil$
- the number of image tiles that must be generated from every buffer θ, computed from eq. (3.3) with $N_{ix} = T_{ix}$ and $N_{iy} = T_{iy}$
- the number of times that each image tile must be read from a buffer $L_{of} = N_{of}/P_{of}$

Given $\zeta = max(\rho, \theta)$, TO performs $L_{of} \cdot \Theta \cdot \zeta$ iterations.

6.5.4 Support to stride-2 convolution

In the case of stride-2 convolution, the activations should be mapped into the four buffers accordingly to the even-odd nature of their (x,y) position in the feature map and not channel-wise. The current behaviour of Tile Organization is invariant of the stride value: the input activations must be reorganized by the host so that they can be correctly interpreted. This software manipulation translates each $T_{ix} \times T_{iy} \times N_{if}$ tile into an equivalent tile with dimensions $T_{ix,2} = T_{ix}/2$, $T_{iy,2} = T_{iy}/2$ and $T_{if,2} = 4 \cdot N_{if}$. A graphical representation of how the activations are fed to Tile Organization can be seen in figure 6.6.

A second version of Tile Organization, with the support to both stride-1 and stride-2 convolution without the need for additional software manipulation was designed as well and its behavioural description was used to derive and test the RTL description. However, the analysis of the RTL showed that the logic utilization and II were not acceptable and would not bring any improvements to the design. Further developments of the accelerator should optimize this description and make the implementation feasible.

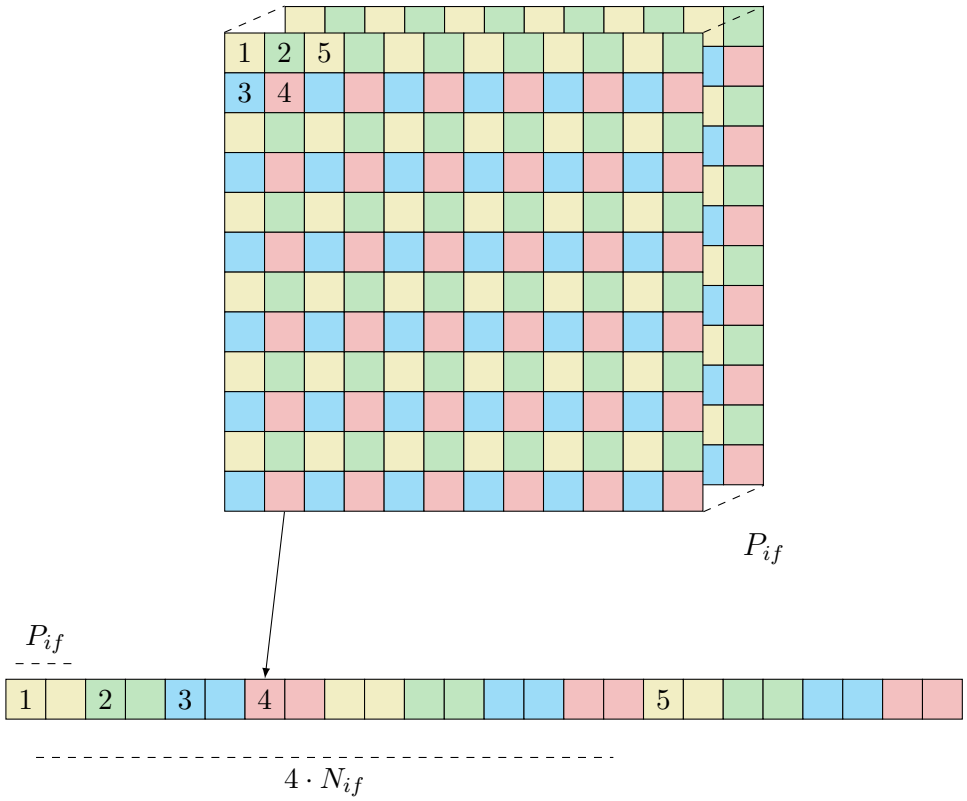


Figure 6.6: Graphical description of the input activation reorganization operated by the host in the case of stride-2 convolution

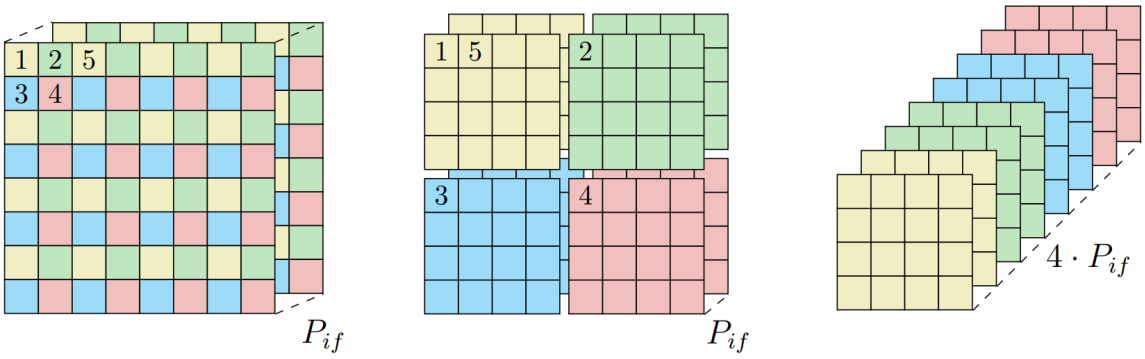


Figure 6.7: Graphical description of the weights reorganization in the case of stride-2 convolution. Kernels characterized by the same color are sent to the same PE.

6.6 Weight Organization

Weight Organization (WO) operates in a similar way to TO: it receives input streams on $512 \cdot P_{if}$ bits that contain $4 \cdot P_{if}$ sets of (4×4) kernels, with 8-bit weights. These weights are stored in four ping-pong buffers organized as

$$[8 \times P_{if} \times P_{of}]\,[4 \times 4]$$

and therefore each with a theoretical memory requirement of $16 \cdot P_{if} \cdot P_{of} \cdot 8$ Byte.

The kernel always receives an even number of streams that can be grouped as couples: for each couple, the first stream contains $4 \cdot P_{if}$ sets of kernels for the convolution with $4 \cdot P_{if}$ consecutive input channels for the computation of the same output channel; the second stream contains a new set of $4 \cdot P_{if}$ kernels that are to be convolved with the same $4 \cdot P_{if}$ input channels but for the computation of a different output channel. Once the buffers are full, WO starts loading the following weights into the second set of buffers while using the values stored in the first set to generate the four output streams. Globally, WO generate four $L_{if} \cdot L_{of} = N_{if}/P_{if} \cdot N_{of}/P_{of}$ sets of $P_{of} \times P_{if} \times 4 \times 4$ outputs with values on 8-bits grouped in $(16 \cdot P_{if} \cdot P_{of})$-bit streams.

6.6.1 Support to stride-2 convolution

In the case of stride-2 convolution, the standard kernel dimension is increased from (4×4) to (8×8). Weight Organization input stream is therefore organized as a $8 \cdot 8 \cdot P_{if}$ filter. As for Tile Organization, also this element supports the organization directed to stride-2 convolution through an additional software manipulation of the data. A graphical representation of the order in which the weights are fed to Weight Organization is shown is figure 6.7

6.7 Host

The host is deployed on the board programmable SoC and its behaviour is described by a Python-based script that relies on the Pynq [28] and Overlay [29] libraries. The code contains all the instructions required to correctly drive the accelerator computation.

The network that the accelerator need to process is described in a *.json* file from which the host can read, for every layer, the kernel size r, the stride s and padding p values, and the feature maps dimensions N_{ix}, N_{iy}, N_{if} and N_{of}. From this data, the code is able to derive the tiling paradigm that best suits the computation and, once the tiling factors T_{ix} and T_{iy} have been chosen, compute all the parameters required to drive the operation.

For every layer, the host flashes the accelerator hardware design on the board, loads the operation parameters on the reserved registers, and then communicates the start of operation signal to WinoAdapt. The input activations and weights are fed to the device through two DMAs whose execution is once again driven by the host program. Finally, once the layer output activations have been computed, the host reads their value from the output DMA, organizes them accordingly to their spatial position into the ofmap and stores them in memory.

Chapter 7

Results

The design described in the previous chapter has been analysed for what concerns

1. Hardware resources utilization
2. Latency estimations
3. Layer execution time
4. Validation of WinoAdapt results through the comparison of with those of the standard convolution

7.1 Hardware resources utilization

The hardware resources utilization has been constantly monitored during the design because it represents the main constrain to the accelerator development. Tables 7.1 and 7.2 report the resources utilization, evaluated for unrolling factors $P_{if} = 2$ and $P_{of} = 2$, both in absolute value and as a percentage of all the available resources, of WinoAdapt and of its main internal components that were introduced in section 6.1, on the reference board, the Xilinx Evaluation Board ZCU104.

Additional resources with respect to those required by the three main components are allocated for the connection between the internal components, and to implement an additional element, called Output Casting, required to properly dimension the output stream. Moreover, two DMAs are responsible to feed and receive data from the device. It is especially evident the large difference between WinoAdapt and Compute Unit CLB Registers utilization with respect to that of their internal components: a large part of these registers are used to implement pipeline and dataflow stages.

Element	CLB	CLB LUTs	LUT as Logic	LUT as Memory	CLB Registers	DSPs	Block RAM	CARRY8
WinoAdapt	27533	164591	155991	8600	161121	935	151.5	11995
Tile Organization	2958	5133	5133	0	8426	51	32	489
Weight Organization	1388	5950	1854	4096	6390	4	0	45
Compute Unit	21087	117245	116833	412	100767	880	72	10405
Processing Element	5091	23803	23704	99	19591	184	18	2397
Input Transformation	757	2504	2504	0	665	0	0	347
Weight Transformation	358	994	994	0	96	0	0	86
Element Wise	2108	5220	5220	0	7786	184	18	685
Output Transformation	2055	8011	8011	0	2431	0	0	1066
Accumulation	1310	6069	6069	0	141	144	0	366

Table 7.1: Resources utilization of WinoAdapt and all its internal components, result of the synthesis for the zcu104 Xilinx Evaluation Board, absolute value.

Element	CLB	CLB LUTs	LUT as Logic	LUT as Memory	CLB Registers	DSPs	Block RAM	CARRY8
WinoAdapt	95.60%	71.44%	67.70%	8.45%	34.97%	54.11%	48.56%	41.65%
Tile Organization	10.27%	2.23%	2.23%	0.00%	1.83%	2.95%	10.26%	1.70%
Weight Organization	4.38%	2.58%	0.80%	4.03%	1.39%	0.23%	0.00%	0.16%
Compute Unit	73.22%	50.89%	50.71%	0.40%	21.87%	50.93%	23.08%	36.13%
Processing Element	17.68%	10.33%	10.29%	0.10%	4.25%	10.65%	5.77%	8.23%
Input Transformation	2.63%	1.09%	1.09%	0.00%	0.14%	0.00%	0.00%	1.30%
Weight Transformation	1.24%	0.43%	0.43%	0.00%	0.02%	0.00%	0.00%	0.30%
Element Wise	7.32%	2.27%	2.27%	0.00%	1.69%	10.65%	5.77%	2.38%
Output Transformation	7.14%	3.48%	3.48%	0.00%	0.53%	0.00%	0.00%	3.70%
Accumulation	4.55%	2.63%	2.63%	0.00%	0.03%	8.33%	0.00%	1.27%

Table 7.2: Resources utilization of WinoAdapt and all its internal components, result of the synthesis for the zcu104 Xilinx Evaluation Board, as a percentage of all the available resources.

Implementation	DSP	LUTs	FF
DSP-based	144 (8%)	12064 (5%)	103 (∼0%)
logic-based	0 (0%)	17392 (8%)	103 (∼0%)

Table 7.3: Vitis HLS resources utilization estimations for a logic-based and a DSP-based implementations for the accumulation operations inside Accumulation. Percentages refer to the Xilinx ZCU104 Evaluation Board resources utilization.

Processing Element

From the results reported in the above cited tables, it is easy to see how the Processing Element requests a large amount of all DSPs, logic and memory space. The analysis of its components resources utilization also highlights many of the compiler design choices.

The components responsible for the input, weights and output transformation all present similar requirements in terms of resources. They don't require any DSP and have only a very limited memory demand in terms of registers. Their most important contribution is that of the LUTs that are required to perform the transformations into and from the Winograd domain. The largest contribution is given by Output Transformation, that operates with a larger bitwidth with respect to Input and Weight Transformation.

Element Wise resources requirements instead strongly differ from those of the other three PE components. Firstly, it has large logic requirements as well, because LUTs are used to implement the adder three that is responsible to accumulates the multiplications results. It also presents a larger number of registers, some of which are used to store the partial products results. Finally, differently from the other three components, EW has important requests both in terms of memory and of DSPs. As was expected from the analysis reported in section 6.3.4 and from equation 6.2, considering the design values $P_{if} = 2$ and $P_{of} = 2$, each PE requires the allocation of 184 DSPs, one for every multiplication that the kernel must perform with the chosen unrolling factors. The RAM memory allocated for this components is instead required to implement the on-chip buffer for weights reutilization.

Accumulation

The Accumulation component performs two different tasks: it is responsible for the accumulation of the four PEs results and for the data bitwidth reduction. While the bitwidth reduction must must be implemented with a purely logic-based approach, since it depends on the current layer parameters and is implemented with a variable bit-shift, two different solutions have been tested for what concerns the accumulation operation: a purely logic-based and a DSP-based implementation. The resources utilization estimated by the synthesis performed by Vitis HLS are reported in table 7.3.

The solution of choice has been the DSP-based one, that was required to reduce the logic utilization and make the design synthesizable on the ZCU104 Evaluation Board. For

Modules	Latency	Interval	Pipelined
Element Wise	6	1	yes
Output Transformation	3	1	yes
Weight Transformation	2	1	yes
Input Transformation	3	1	yes
Accumulation	2	1	yes
Tile Organization	49	26	yes
Weight Organization	2	1	yes
Output Casting	3	1	yes

Table 7.4: Iteration interval and iteration latency of the accelerator components.

this implementation, the adder three that performs the accumulation of the four results requires the allocation of additional $2 \cdot P_{of} \cdot 36 = 144$ DSPs, that compress the four values into two, and then some additional logic to accumulate the two partial results.

Compute Unit

The large resources utilization of the four PEs and of Accumulation is obviously reflected also in the Compute Unit, which makes up for the better part of the accelerator LUT-, DSP- and logic-utilization.

Tile Organization

The main contribution of Tile Organization with respect to the overall design requirements is, as expected, in terms of memory: in order to be fast and flexible, the kernel requires the implementation of a buffer that can be simultaneously accessed from different addresses for both read and write operations. This translates into a slight overhead of the available memory blocks. Indeed, the four buffers have a theoretical memory demand of 115.200 Bite or 921 kb (as reported from equation 6.5), while the synthesis performed by Vivado requires the allocation of 1.21Mb of memory. The design has been strongly optimized so to reduce the logic utilization.

Weight Organization

Weight Organization presents a structure similar to that of Tile Organization. However, the buffer in WO is implemented with a purely register-based approach, and does not use any block of RAM. This allows to achieve a faster execution and reduce the logic utilization with respect to a RAM-based implementation. Each PE shows to require the allocation of 635kb of memory.

Layer	N_{ix}	N_{iy}	N_{if}	N_{of}	r	s	p
Layer 1	14	14	128	128	3	1	0
Layer 2	30	30	64	64	3	1	0
Layer 3	30	30	128	128	3	1	0
Layer 4	54	54	32	32	3	1	0
ResNet 1	56	56	64	64	3	1	1
ResNet 2	28	28	128	128	3	1	3
ResNet 3	14	14	256	256	3	1	2
ResNet 4	7	7	512	512	3	1	2

Table 7.5: Reference layers used for the evaluation of the accelerator behaviour

7.2 Latency estimations

The second parameters that have driven the accelerator design were the components Iteration Interval (II) and Iteration Latency, respectively the number of cycles that pass between the start of two contiguous iterations and the number of cycles required to complete each iteration. The Iteration Interval can be modified through the components optimization, that has a target II = 1. Each component latency values were estimated through the synthesis performed by Vitis HLS and are reported in table 7.4.

The only component that was not possible to optimize in terms of iteration interval is Tile Organization, that presents an II = 26. As will be highlighted also later, this large Iteration Interval will have a strong impact on the global accelerator behaviour in terms of execution time, because the correct organization of the activations will require a time longer than the one required by the Winograd convolution computation. This long iteration interval is due to the impossibility of reading and writing the content of multiple RAM cells at the same time. Future development of this project should try to further reduce the II, possibly also by exploring completely new designs.

7.3 Execution time

WinoAdapt and its components were characterized in terms of execution time to understand their relative impact on the complete accelerator behaviour. The behaviour has been evaluated for the reference layers reported in table 7.5 and, when applicable, for different scaling factors.

7.3.1 Tile Organization

Tile Organization is the component that presents the largest Iteration Interval and Iteration Latency values ans is therefore expected to bring the major contribution to the overall

Layer	T_{ix}	T_{iy}	Θ	ρ	θ	L_{of}	$L_{of} \cdot \Theta \cdot \zeta$	Execution Time
Layer 1	14	14	1	196	144	64	12544	7.3088 ms
Layer 2	30	30	1	450	392	32	14400	8.2616 ms
Layer 2	14	14	9	98	72	32	28224	8.9268 ms
Layer 4	30	30	4	225	196	16	14400	10.1661 ms
Layer 4	14	14	25	49	36	16	19600	8.4306 ms

Table 7.6: Execution time of Tile Organization on the ZCU104 board. The reference layers have been tested with different tiling factors and the reported execution time has been obtained as the average of four executions. The definition of the reported parameters can be recalled in section 6.5.3.

Layer	T_{ix}	T_{iy}	Θ	ρ	θ	L_{of}	$L_{of} \cdot \Theta \cdot \zeta$	Execution time
ResNet 1	30	30	4	450	392	32	57600	38.2458 ms
ResNet 1	22	22	9	242	200	32	69696	35.0316 ms
ResNet 1	14	14	25	98	72	32	78400	30.9667 ms
ResNet 1	10	10	49	50	32	32	78400	42.8266 ms
ResNet 1	10	30	14	150	112	32	67200	39.4479 ms
ResNet 1	10	58	7	290	1568	32	351232	51.4269 ms
ResNet 1	14	58	5	406	1568	32	250880	44.6196 ms

Table 7.7: Execution time of ResNet 1 layer, ZCU104 board, with different tiling factors.

accelerator execution time. The execution time, computed as the average of four successive executions for each layer, is reported in table 7.6 along with the loop parameters that Tile Organization receives to correctly perform the computation. The comparison between the execution time and the component global number of iteration, that noticeably increases with the tiling, shows how tiling is indeed effective at improving the latency, since the execution time does not increase with the number of iterations and, in the case of the higher tiling (i.e. smaller tiling factors) applied to layer 4, effectively reduces the execution time.

Table 7.7 reports the execution times of Tile Organization for an input feature map with dimensions $N_{ix} = 56$, $N_{iy} = 56$, $N_{if} = 64$, with $N_{of} = 64$ and for different tiling factors. This experiments show how it is also possible to use tiling factors $T_{ix} \neq T_{iy}$, but they appear to be less efficient than the case of $T_{ix} = T_{iy}$, especially if the difference between the two tiling factors increases. A similar analysis has been done also for the other ResNet layers, but their results do not provide additional information.

The reported values are still useful to understand how TO behaves with different layers and different tiling factors. However, the time characterization of the component will be shown not to be accurate, because in order to obtain the characterization TO had to be

Layer	T_{ix}	T_{iy}	WinoAdapt	2-PE CU	PE
layer 1	14	14	4.9942 ms	-	7.6652 ms
layer 2	30	30	7.8195 ms	4.9905 ms	8.6632 ms
layer 2	18	18	7.0983 ms	-	-
layer 2	10	10	7.4184 ms	-	-
layer 3	30	30	-	17.5572 ms	31.0236 ms
layer 3	18	18	27.2871 ms	-	-
layer 3	14	14	33.0108 ms	-	-
layer 4	54	54	-	5.5618 ms	9.3278 ms
layer 4	30	30	5.0437 ms	-	-
layer 4	18	18	6.1040 ms	-	-
layer 4	14	14	5.6471 ms	-	-
layer 4	10	10	5.6535 ms	-	-

Table 7.8: Execution time of the reference layers on the ZCU104 board with different tiling factors. CU and PE execution times have been evaluated only for the case of $T_{ix} = N_{ix}$ and $T_{iy} = N_{iy}$.

coupled with another component, called Output Casting. Output Casting is responsible for resizing the output streams and expand them from $36 \cdot P_{of} \cdot 8 = 576$ bits (which is the output stream dimension with the current unrolling factor $P_{of} = 2$) to 1024 bits, that is the closer power of 2's: this element is required to implement an acceptable interface between TO and the four DMAs that collect its outputs.

7.3.2 WinoAdapt

In order to get a comprehensive view of the behaviour of the accelerator, its performance characterization involves two different analysis. The first step has seen the evaluation of the layers execution time considering different tiling factors. The second analysis involved evaluating the global execution time of the succession of all the considered layers.

Single layer execution

The execution of the reference layers, with the same tiling factors already used for the characterization of TO, are reported in table 7.8 along with the execution time of the 2-PE CU and stand-alone PE. While these results clearly show how doubling the number of PEs leads to an improvement in terms of throughput, an equivalent gain is not evident for what concerns the full accelerator and its 4-PE Compute Unit. Indeed, the full layer execution time of WinoAdapt is often significantly higher than that of the 2-PE CU. However, the main difference between the two design is the presence of two additional components,

Layer	Execution time
Global	25.5387 s
Layer 1	4.9942 ms
Layer 2	7.8195 ms
Layer 3	27.2871 ms
Layer 4	6.8349 ms

Layer	Execution time
Global	21.1610 s
ResNet 1	19.1439 ms
ResNet 2	27.3502 ms
ResNet 3	11.1464 ms
ResNet 4	6.9010 ms

Table 7.9: Execution time of multiple layers on the ZCU104 Xilinx Evaluation Board.

namely TO and WO, in front of the compute unit, which makes the full execution of WinoAdapt longer, especially because of the long II that characterizes TO.

Moreover, during the time evaluation of the processing element the image tiles were fed directly to the DMAs through a process driven by the host. Finally, for larger layers, as the considered layer 4, WinoAdapt already presents a better performance with respect to the 2-PE CU, showing that the implemented activations reuse paradigm is indeed effective. Despite the lack of improvements in the execution time, WinoAdapt is still a better solution than the stand-alone PE or 2-PE Compute Unit, because its design brings the additional advantage of the support to stride-2 convolution.

Empty cells in the table highlight how the tiling factors cannot be arbitrary and some values, like $T_{ix} = 54$ and $T_{iy} = 54$ for Layer 4 cannot be used because the tile would exceed the on-chip buffer dimension.

Multiple layers execution

The results of the execution of all the test layers in sequence is reported in table 7.9. This final characterization has been retrieved programming the host with a slight different program than before. In this final version of the code, the host is responsible to read the layer parameters from a *.json* file, determine the appropriate tiling factors, compute the execution parameters required from the accelerator, and feed the input activations and weights to WinoAdapt accordingly to the desired tiling paradigm. The time required for the complete execution of both sets of layer is significantly higher than the single layers execution times, because they keep track of the time required for the host code execution as well as that for the Winograd convolution.

7.4 Validation of the hardware accelerator results

The designed hardware accelerator results have been validated through the description of a Python-based model of the hardware device. The model is able to perfectly match the results computed by the accelerator and can be used to compare them with the result of the quantized standard convolution algorithm already described in section 3.1.

Layer	Division Factor (D)	$log_2(D)$	Bit-shift
Layer 1	7377.70	12.8	13
Layer 2	4789.06	12.2	13
Layer 3	8413.75	13.1	14
Layer 4	3175.92	11.6	12
ResNet 1	5221.78	12.4	13
ResNet 2	9071.25	13.1	14
ResNet 3	7483.97	12.9	13
ResNet 4	6553.03	12.7	13

Table 7.10: Scaling factors used to verify WinoAdapt output error.

The device has been validated also for what concerns stride-2 convolution, proving that the decomposition method described in section 5.3 coupled with the four PEs architecture can be used to accelerate stride-2 convolution through the Winograd algorithm.

7.4.1 Scaling paradigm

Data bitwidth reduction can be implemented with either a division or a bit-shift with a given *scaling factor*, that should theoretically be derived from the ofmap largest value as

```
scaling factor = max(ofmap)/127
```

where the division by 127 derives from the fact that the data is stored as signed integers on 8 bits. Obviously, it is not possible to dynamically derive this value during execution, so the scaling factor has been computed before the inference time. We have therefore used the accelerator model to compute, for every test layer, the convolution between 10 different sets of input activations and weights ans asked the model to provide the not-scaled results so be able to identify the maximum out of every ofmap.

The scaling factors have been stored in a table that the host can access once it has read the current layer parameters and are reported in table 7.10. WinoAdapt performs the scaling operation through a bit-shift.

7.4.2 Result accuracy

WinoAdapt results have been validated by comparing their values with those of the standard quantized convolution output. The introduced numerical error in every layer appears to be higher than complex Winograd theoretical one, reported in table 3.1, but nevertheless better than the one that is theoretically possible to obtain with the standard Winograd convolution. These results could be improved in the bitwidth reduction was implemented through a division instead than with a bit-shift: for reference, in the case of the ResNet 1

Layer	Bit-shift value	Ops ratio	Max difference	Error Mean	Error standard deviation
Layer 4	13	2.9108	46	4.3	11.6673
Layer 1	13	2.3967	42	9.3	8.9596
Layer 2	14	2.7513	63	3.3	15.3353
Layer 3	12	2.7514	77	2.5	18.5009
ResNet 1	13	3.1304	75	4.3	14.9383
ResNet 2	14	3.1304	86	2.3	15.8854
ResNet 3	13	3.1304	62	7.3	9.2974
ResNet 4	13	1.7608	66	5.5	10.7537

Table 7.11: Analysis of the error introduced by WinoAdapt during the convolution computation with respect to the 8-bit quantized standard convolution output.

layer, this would reduce the maximum difference to 45 and the standard deviation to 6.64. This second implementation does however require significantly more logic.

Chapter 8

Conclusions

This work described the design principles of WinoAdapt and reports the analysis of its behaviour in terms of logic utilization, layer execution time and introduced error.

The analysis has proven that the `F(4,3)` complex Winograd algorithm can effectively be exploited to achieve the same acceleration provided by the standard Winograd convolution with the request of minimal additional resources, while also improving the computation prediction accuracy by reducing the error introduced during the computation. The matrix reuse paradigm described is section 5.2 and the decomposition method described in section 5.3 allow to increase the device flexibility in terms of both supported filter size and stride with the request of minimal resources overhead. Finally, the buffer designed to support ifmap tiling shows the desired flexibility in terms of variable input tiles dimensions.

The time characterization has shown that the designed accelerator presents no performance degradation with respect to the Processing Element execution, and often effectively accelerates the execution. At the same time, the analysis of the introduced error has shown a significant improvements with respect to the case of standard Winograd.

A further development of the project should focus on the optimization of the device elements that have already been designed, with the primary target of logic utilization reduction. This step is necessary because the current design already requires the allocation of all the LUTs available on the target FPGA. Additionally, Tile Organization requires a strong optimization in terms of iteration interval, that must be reduced to improve the design global latency. The accelerator output accuracy can be increased as well by implementing the bitwidth reduction through a floating point division instead of the current bit-shift. Support to normalization, non-linear and pooling layers should be implemented as well.

Finally, the efficiency in the execution of stride-2 layers can be improved by designing new Tile Organization and Weight Organization elements that are able to discriminate between the cases of stride-1 and stride-2 convolutions, so to be able to autonomously organize the input activations in both cases. This would remove the requirement of software

data manipulation of the input activations. Since the design is flashed on the board at each iteration, one possibility would be the implementation of two separate designs, one stride-1 oriented and one stride-2 oriented, if the logic requirements should not allow to achieve the desired flexibility with just one design.

References

[1] V. Sze, Y.-H. Chen, Tien-JuYang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017. DOI: 10.1109/JPROC.2017.2761740.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015. DOI: 10.1145/2775054.2694358. [Online]. Available: https://doi.org/10.1038/nature14539.

[3] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," 2017. DOI: https://doi.org/10.48550/arXiv.1611.03530. arXiv: 1611.03530 [cs.LG].

[4] A. L. Maas, A. Y. Hannung, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," 2013.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2013. DOI: 10.48550/ARXIV.1311.2524. [Online]. Available: https://arxiv.org/abs/1311.2524.

[6] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," vol. 27, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/00ec53c4682d36f5c4359f4ae7bd7ba1-Paper.pdf.

[7] M. R. Vemparala, A. Frickenstein, and W. Stechele, "An efficient fpga accelerator design for optimized cnns using opencl," M. Schoeberl, C. Hochberger, S. Uhrig, J. Brehm, and T. Pionteck, Eds., pp. 236–249, DOI: 10.1007/978-3-030-18656-2_18.

[8] C. Yang, Y. Wang, X. Wang, and L. Geng, "Wra: A 2.2-to-6.3 tops highly unified dynamically reconfigurable accelerator using a novel winograd decomposition algorithm for convolutional neural networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 9, pp. 3408–3493, 2019. DOI: 10.1109/TCSI.2019.292868.

[9] Y. Ren and X. Cheng, "Review of convolutional neural network optimization and training in image processing," in *Tenth International Symposium on Precision Engineering Measurements and Instrumentation*, J. Tan and J. Lin, Eds., International Society for Optics and Photonics, vol. 11053, SPIE, 2019, p. 1 105 331. DOI: 10.1117/12.2512087. [Online]. Available: https://doi.org/10.1117/12.2512087.

[10] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018. DOI: 10.1109/TVLSI.2018.2815603.

[11] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," pp. 10–14, 2014. DOI: 10.1109/ISSCC.2014.6757323.

[12] T. P. Morgan. "Nvidia pushes deep learning inference with new pascal gpus." (2016), [Online]. Available: https://www.nextplatform.com/2016/09/13/nvidia-pushes-deep-learning-inference-new-pascal-gpus/ (visited on 03/20/2023).

[13] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," S. Wermter, C. Weber, W. Duch, *et al.*, Eds., pp. 281–290, 2014.

[14] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. arXiv: 1509.09308. [Online]. Available: http://arxiv.org/abs/1509.09308.

[15] D. Liu, T. Chen, S. Liu, *et al.*, "Pudiannao: A polyvalent machine learning accelerator," *SIGPLAN Not.*, vol. 50, no. 4, pp. 369–381, Mar. 2015, ISSN: 0362-1340. DOI: 10.1145/2775054.2694358. [Online]. Available: https://doi.org/10.1145/2775054.2694358.

[16] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, 2016.

[17] S. Han, X. Liu, H. Mao, *et al.*, "Eie: Efficient inference engine on compressed deep neural network," 2016. arXiv: 1602.01528 [cs.CV].

[18] Ó. Lucía, E. Monmasson, D. Navarro, L. A. Barragán, I. Urriza, and J. I. Artigas, "Chapter 29 - modern control architectures and implementation," in *Control of Power Electronic Converters and Systems*, F. Blaabjerg, Ed., Academic Press, 2018, pp. 477–502, ISBN: 978-0-12-816136-4. DOI: https://doi.org/10.1016/B978-0-12-816136-4.00030-0.

[19] Y. Sun, G. Wang, B. Yin, J. R. Cavallaro, and T. Ly, "Chapter 8 - high-level design tools for complex dsp applications," in *DSP for Embedded and Real-Time Systems*, R. Oshana, Ed., Oxford: Newnes, 2012, pp. 133–155, ISBN: 978-0-12-386535-9. DOI: https://doi.org/10.1016/B978-0-12-386535-9.00008-1. [On-

line]. Available: https://www.sciencedirect.com/science/article/pii/B9780123865359000081.

[20] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980.

[21] X. Liu, Y. Chen, C. Hao, A. Dhar, and D. Chen, "Winocnn: Kernel sharing winograd systolic array for efficient convolutional neural network acceleration on fpgas," *CoRR*, vol. abs/ 2107.04244, 2021. arXiv: 2107.04244. [Online]. Available: https://arxiv.org/abs/2107.04244.

[22] L. Meng and J. Brothers, "Efficient winograd convolution via integer arithmetic," *CoRR*, vol. abs/1901.01965, 2019. DOI: 10.48550/arXiv.1901.01965. arXiv: 1901.01965. [Online]. Available: http://arxiv.org/abs/1901.01965.

[23] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV].

[24] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170, ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. [Online]. Available: https://doi.org/10.1145/2684746.2689060.

[25] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, *Deep learning with int8 optimization on xilinx devices*, White Paper: UltraScale and UltraScale+ FPGAs, 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8.

[26] *Zcu104 evaluation board*, English, version Version 1.1, XILINX, 92 pp., 2018-10-9.

[27] *Vitis high-level synthesis user guide*, English, version Version v2022.2, XILINX, 726 pp., 2022-12-07.

[28] "Pynq introduction." (), [Online]. Available: https://pynq.readthedocs.io/en/latest/ (visited on 04/04/2023).

[29] "Pynq overlays." (), [Online]. Available: https://pynq.readthedocs.io/en/latest/pynq_overlays.html (visited on 04/04/2023).