Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Design and Optimization of a Winograd aware IP for Quantized Neural Networks

Supervisor
Prof. Claudio PASSERONE

Candidate
Davide LEZZOCHE

Advisor
Pierpaolo MORÌ

April 2022

Abstract

Convolutional Neural Networks (CNNs) are increasingly used in the field of deep learning. Among their possible applications are computer vision, speech recognition, and image classification. Nowadays, CNNs have reached very high levels of precision, at the cost of a huge amount of multiplications to perform and parameters to store. GPUs are very popular in the field of CNNs since they are characterized by excellent computing performance. However, their excessive power consumption does not make them the best choice for embedded applications. Instead, FPGAs represent a good compromise between throughput, flexibility, reconfigurability, and energy efficiency. Given the limited resources of FPGAs and the large computational cost and storage demand (both on-chip and off-chip) of CNNs, several optimizations are required to implement an FPGA-based CNN accelerator. Quantization, loop unrolling, and data vectorization help reduce resources demand and speeds up inference. The Winograd algorithm can be used to reduce the computational cost as well, it greatly decreases the number of multiplications required to perform a convolution, however, it introduces a numerical error that is accentuated by quantization. Methods, like residue number system (RNS) representation and the introduction of the complex numbers field, that avoid or limit this error have been introduced in the literature. This thesis analyzes and compares the possible Winograd representations for the $F(2 \times 2, 3 \times 3)$, $F(4 \times 4, 3 \times 3)$ and $F(6 \times 6, 3 \times 3)$, to be used in an 8-bit fixed point quantized neural network. From the comparison, the complex representation of $F(4 \times 4, 3 \times 3)$ results in the best compromise between complexity reduction and the numerical error introduced. Therefore, it is implemented in a deeply pipelined IP aware of both the standard and Winograd convolutions. The results achieved by the test on the latencies of the two algorithms shows that the Winograd convolution outperforms the standard one. However, it is also demonstrated that the Winograd awareness greatly affects the logic resources demand, limiting the maximum level reachable by the parallelization.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

Recent advances in convolutional neural networks (CNNs) [1] make them increasingly used in the field of artificial intelligence (AI) for different applications such as image classification [2], video analysis [3] and speech recognition [4].

Nowadays, FPGAs are becoming very popular in the CNNs field, especially on embedded applications [5]. They can guarantee excellent flexibility and good power efficiency thanks to programmable logic blocks and interconnections. However, FPGAs have limited resources availability concerning computational hardware and on-chip memory. The high performance reached by modern CNNs requires performing a huge number of calculations and storing parameters. Therefore, computational and architectural optimizations are needed when implementing CCNs on FPGAs.

Winograd's algorithm [6] is a computational optimization that makes it possible to greatly reduce the multiplications required to perform a convolution [7], at the cost of introducing a numerical error, unless a high-precision value's representation is used. A low-precision fixed-point quantization can bring benefits to a neural network such as a reduced memory footprint and faster inference [8] and it becomes essential in a resource-constrained device, such as FPGAs. The possibility to combine the Winograd algorithm with a low-precision fixed-point quantization was introduced by [9] and [10]. To achieve this, they have exploited representations to replace the standard real number system with the complex field or Residue Number System (RNS).

This thesis analyzes these two possible methods, to design an 8-bit quantized Winograd aware IP for a convolutional neural network, to be implemented on a XILINX FPGA.

1.2 Organization

This document is structured in 8 chapters.

- The second chapter summarizes the basic knowledge of the CNNs and introduces the theory behind the used optimization in the IP design.
- The third chapter is dedicated to the works that have represented the starting point of this project.
- The fourth chapter analyzes the possible implementations of the Winograd algorithm. The pros and cons of each are explained and then compared.
- The fifth chapter delves into the optimizations made. It explains how they were applied in the design and what benefits they brought.
- The sixth chapter explains in detail how the project design is structured. It also shows some relevant code examples and comments on them.
- The seventh chapter shows the results obtained from the implementations of the designed system, using graphs and tables that collect the results of the simulations and synthesis.
- The eighth chapter draws conclusions from this work and proposes possible future developments.

Chapter 2

Background

This chapter shows the basic knowledge of neural networks and, in particular, the convolutional ones (CNNs) whose structure is illustrated. The possible platforms that can be used to implement CNNs are also listed. In the end, the Winograd algorithm is shown: it will be the basis of computational optimizations carried out in this project.

2.1 Neural networks

Neural networks are part of the artificial intelligence (AI) branch inspired by the human brain. They use machine learning algorithms to perform their tasks, they learn how to handle a problem through the *training* process [11].

The basic building block of neural networks is the neuron, like in the human brain. The artificial neuron performs a weighted sum of different inputs. Its function is represented in figure 2.1, where:

- X_i are the inputs
- W_i are named *weights* and each of them multiply its connected inputs
- Σ is the sum operator
- b is a *bias* that is possibly added
- $f()$ is a non-linear function, also called *activation function*, and filters outputs under a certain threshold
- Y is the output also called *activation*

In a neural network, neurons are grouped in layers. Figure 2.2 shows an example, where a first layer receives the inputs and propagates them to a middle layer also

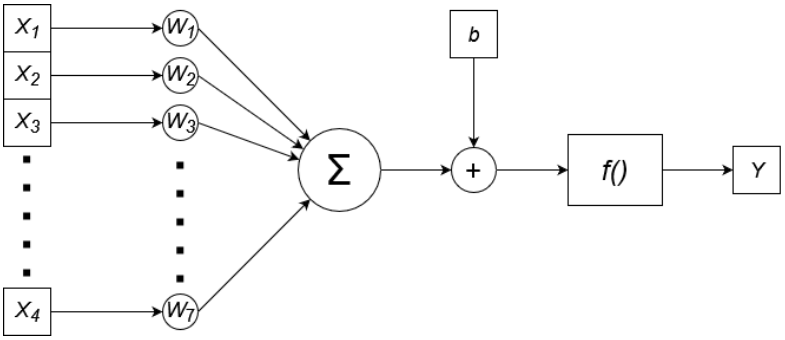


Figure 2.1: Artificial Neuron representation

called *"hidden"* layer. Neurons in the hidden layer perform their weighted sums and then propagate their activations to the last layer, which presents the outputs. Neural networks with more than one hidden layer are considered Deep Neural Networks (DNNs).

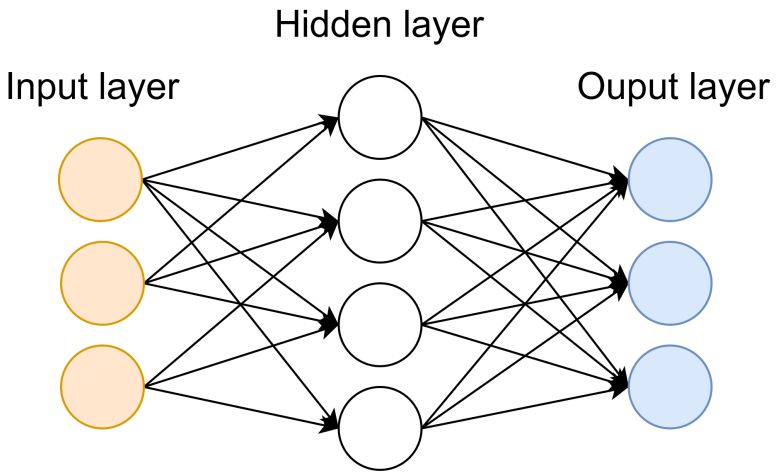


Figure 2.2: Neural networks layered structure

2.2 Convolutional neural networks

Convolutional neural networks are deep neural networks designed for processing structured data arrays, such as images. Their structure can be composed of five different types of layers: convolutional, pooling, nonlinearity, normalization and fully connected. These layers alternate between them and each one applies a mathematical function to their inputs and propagates their outputs to the next layer.

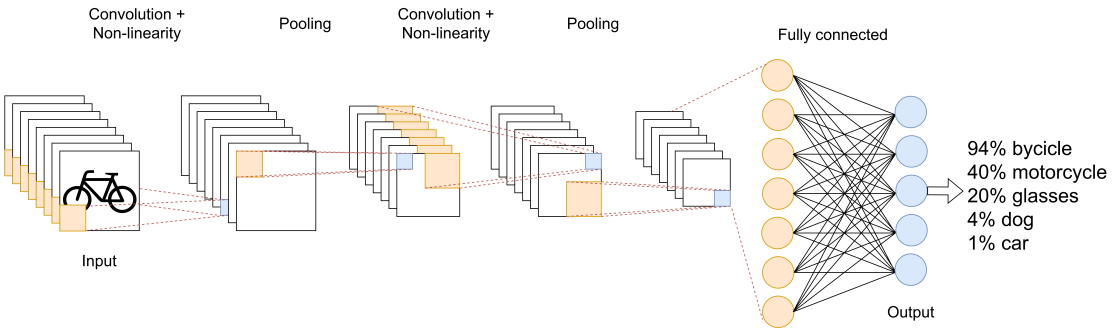


Figure 2.3: Structure of a CNN for image classification

2.2.1 Convolutional layer

The convolutional layer is in charge of performing the convolution operation, to extract features from its inputs. A Convolution is a sparse linear transformation, sparse because few inputs influence an output. Considering a two-dimensional convolution, as in figure 2.4, the operation consists of sliding a *kernel* across an input, also called input feature map (*IFMap*). At each position assumed by the kernel, overlapped elements of the two operands are multiplied and the products are summed, to result in an output element. The final output (OFMap) is a feature map composed of the output of all the positions assumed by the kernel.

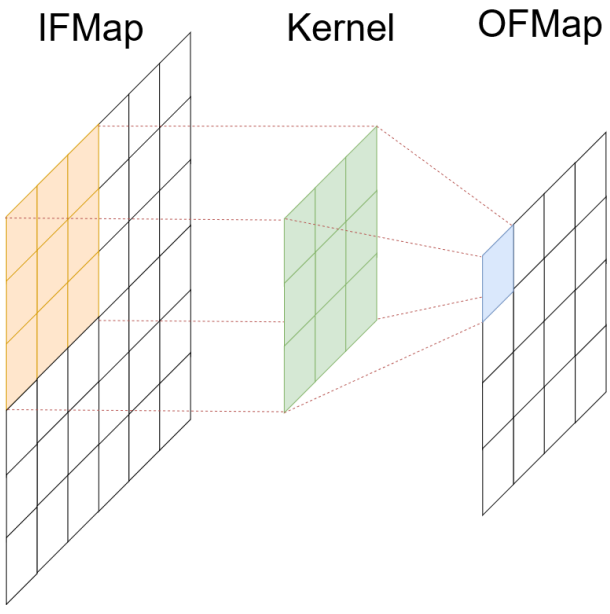


Figure 2.4: Representation of a kernel application in a 2D convolution

The output dimensions are evaluated by equation 2.1, which is valid for both x and y dimensions and where:

- N_o is the output dimension
- N_i is the input dimension
- N_k is the kernel dimension
- s is the value of the *stride*
- p is the value of the *zero padding*

$$N_o = \lfloor \frac{N_i + 2p - N_k}{s} \rfloor + 1 \tag{2.1}$$

The stride of a convolution is defined as the distance between two successive positions of the kernel. Zero padding, instead, is a technique that consists of padding with zeros on the edges of the IFMap, and the value p is referred to as the thickness of these edges. This procedure is often used to obtain an OFMap of the same dimensions of the IFMap using p equal to the lower half of the kernel dimension (with $s = 1$).

Into the convolutional layer of a CNN the operation is performed between multidimensional operands (figure 2.5):

- *IFMaps*: the input is composed of a set of N_{if} IFMaps
- *weights*: sets of N_{if} kernels are grouped in filters, the weights are composed of N_{of} filters

To each input feature map corresponds an *input channel*, and to each output feature map corresponds an *output channel*. The multidimensional convolution works similarly to the 2D case. Each 3D filter slides on the IFMaps, in each assumed position the overlapped values between the filter and IFMaps elements are multiplied and their products summed. An OFMap is produced at the end of a filter application. The output of the convolution is a set of N_{of} OFMaps, one for each filter application.

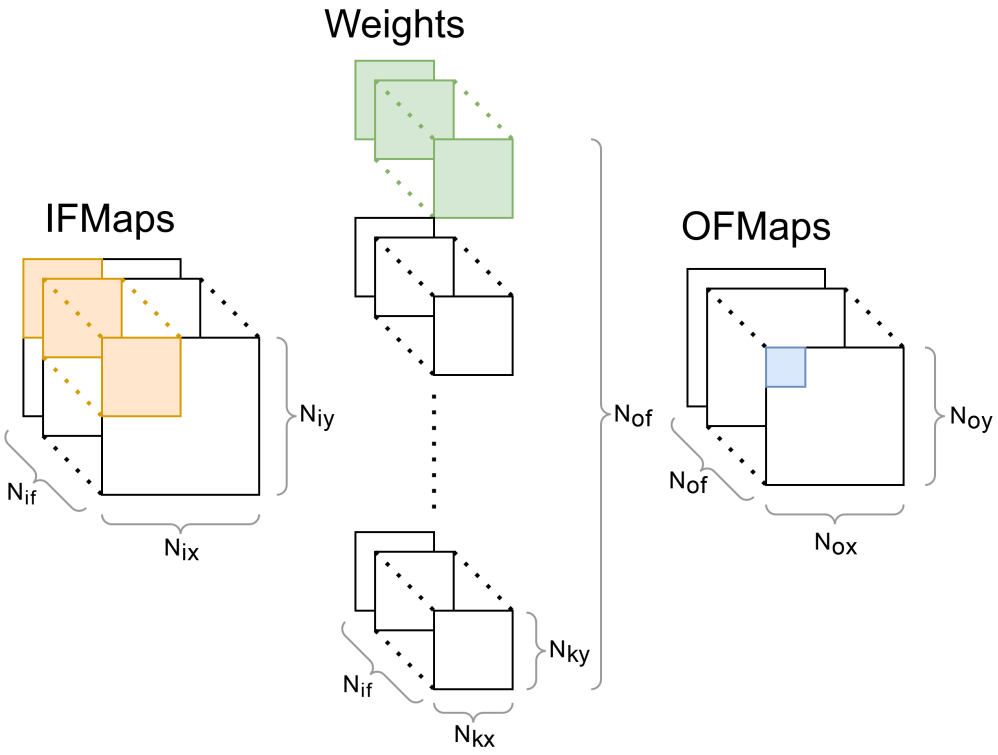


Figure 2.5: Representation of a multidimensional convolution

Summarizing a convolution consists of a MAC operation (multiply and accumulate) nested in a six-dimensional loop, as shown by algorithm 2. The total number of MACs required is given by the product:

$$N_{ox} \times N_{oy} \times N_{of} \times N_{kx} \times N_{ky} \times N_{if} \tag{2.2}$$

.

Algorithm 1 6D nested loop of a convolution

for $oy = 0;\ oy < N_{oy};\ oy++$ **do**
  for $ox = 0;\ ox < N_{ox};\ ox++$ **do**
    for $of = 0;\ of < N_{of};\ of++$ **do**
      for $if = 0;\ if < N_{if};\ if++$ **do**
        for $ky = 0;\ ky < N_{ky};\ ky++$ **do**
          for $kx = 0;\ kx < N_{kx};\ kx++$ **do**
            OFMaps(ox, oy, of)+=
            Weights(kx, ky, if, of) $\times IFMaps(s \times ox + kx, s \times oy + ky, if)$
            end
          end
        end
      end
    end
  end

2.2.2 Pooling layers

Pooling layers can be used in a neural network to reduce the size of feature maps. The layer works by dividing the input into tiles, typically not overlapped, and summarizing them using a function. The most common used functions are max and average pooling:

- *max pooling*: the input tiles are substituted by their maximum value
- *average pooling*: the input tiles are substituted by the average of their values

Their examples are shown in figure 2.6, where it is visible the size reduction: in both two cases from a 4×4 input is derived a 2×2 output.

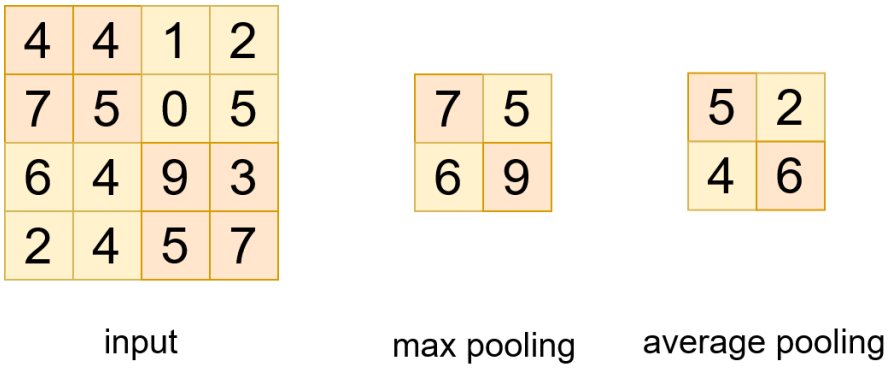


Figure 2.6: max and average pooling examples

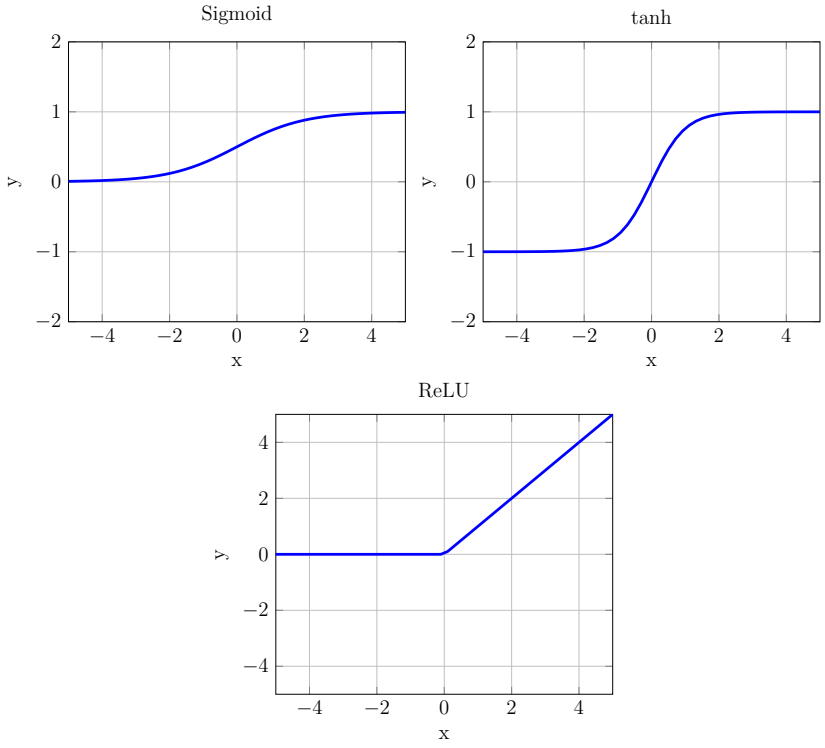
2.2.3 Non-linearity layers

Non-linearity layers apply a nonlinear function to all the elements of the previous layer output feature maps, resulting in an output of the same dimension. They usually are applied after convolutional or fully connected layers.

Commonly applied nonlinear functions are :

- rectified linear unit $(ReLU)$: $y = max(0, x)$
- sigmoid: $y = \frac{1}{1+e^{-x}}$;
- hyperbolic tangent $(tanh)$: $y = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$;

The main advantage of the sigmoid function is that its output range is 0 to 1. It can be used for example to predict a probability. The tanh function ranges between -1 and 1 and its main use is the classification between two classes. The ReLU function is the faster function because it doesn't involve a mathematical operation: it only sets to 0 the negative values and, for this reason, it is widely used in CNN. In addition, for training purposes, the ReLU outperforms the other two functions.



2.2.4 Normalization layers

Their purpose is to accelerate the training process and to improve the net accuracy, by normalizing the input data of the next layer. Batch normalization layers are the most used. Their application results in an output with a mean (σ) equal to 0 and a standard deviation(μ) equal to 1. After batch normalization, the output is scaled and shifted using parameters adjusted from training (γ and β). Equation 2.3 is used to derive the output y from an input x. To avoid dividing by zero, a constant (ϵ) is added to the denominator.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2.3}$$

2.2.5 Fully connected layers

CNNs analyze and abstract the content of the starting set of feature maps in a multilayered sequence of convolutional, non-linear, pooling layers and normalization layers. At the end of a CNN, a set of fully connected layers (FCs) is usually used to achieve a final classification.

Fully connected layers (FCs) apply a linear transformation, as in the case of convolutional layers. In this case, however, all the inputs influence all the outputs, they are connected in a structure as the one illustrated in figure 2.7. In this layer, the inputs are flattened before starting the kernel application ad the two operands are of the same dimension.

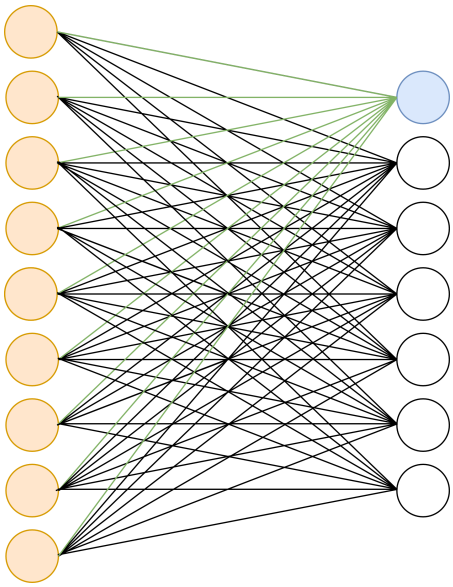


Figure 2.7: Dots representation of a fully connected layer

2.2.6 Accuracy and data sets

In the image classification field, the common parameter used to evaluate the performance of a neural network in terms of correct predictions is the accuracy. The accuracy is defined by the following formula:

$$Accuracy = \frac{number\ of\ correct\ predictions}{total\ number\ of\ predictions} \tag{2.4}$$

When comparing two different network models it is important to refer to the same set of inputs. At this scope, different popular data sets can be used. For example, some of the most popular data sets for image classification are MNIST [12] and ImageNet [13].

MNIST is composed of 70000 images (10000 are used for testing and the others are used for the training task) of 28×28 handwritten digits, so there are 10 possible classes.

ImageNet is composed of 1.45 million RGB images of 256×256 pixels, divided in 1000 classes. 100 test images are provided for each class, the others are used for training and validation purposes.

2.3 Hardware platforms

The complexity in terms of the number of parameters and operations reached by modern CNNs makes it crucial to choose the hardware device to implement them. The various possibilities such as ASICs, CPUs, GPUs, and FPGAs offer different advantages at the cost of different trade-off. This section aims to analyze the four possibilities to find the best solution to accelerate the inference of a CNN on an embedded device.

CPUs

Central processing units (CPUs) present a limited number of cores and are designed to perform, in a sequential manner, a set of instructions. They are non-optimized for parallelism. AI algorithms require parallel computations, performances on CPUs are therefore limited. Being CPUs designed for general purpose, their hardware offers great flexibility but at the cost of high power consumption.

GPUs

Graphics processing units (GPUs) are the most used accelerators. Their SIMD-based architecture makes them the most effective in parallel computations, such as those required by CNNs. Their biggest drawback, however, is their high power consumption.

ASIC

Application-Specific Integrated Circuits (ASICs) are highly optimized devices to execute their function. ASICs have a long production cycle and are difficult to program. Their hardware allows for achieving very good latency performance and minimal power consumption, at the cost of sacrificing flexibility. An ASIC, in fact, can only perform the task for which it is designed. For these reasons, their major field is a massive production.

FPGA

Field Programmable Gate Arrays (FPGAs) are programmable logic devices structured as matrices of Configurable Logic Blocks (CLBs) connected by programmable interconnections (see Figure 2.8), that allow excellent flexibility. They can be easily programmed to solve specific functions and optimized for low power consumption.

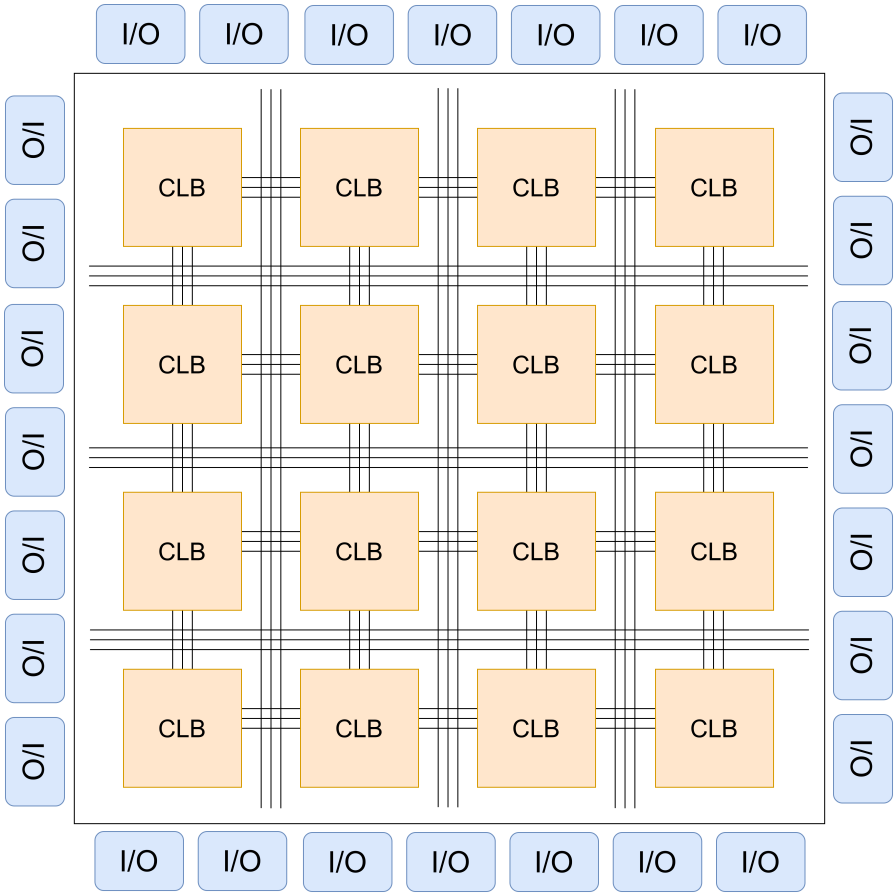


Figure 2.8: FPGA structure is composed of configurable logic and programmable interconnections

Comparison

Table 2.1 summarizes the comparison between the possible platforms. CPUs result as inefficient to accelerating a CNN. Since an embedded application requires low power consumption also a GPU cannot be used. Due to its higher flexibility, FPGA is preferred over an ASIC solution. Considering all this, the platform chosen to accelerate the project of this thesis is an FPGA, in particular, XILINX Zynq UltraScale+ MPSoC ZCU104 (its features are shown in table 2.2).

Platform	throughput	power consumption	flexibility
CPUs	low	very high	very high
GPUs	very high	very high	high
ASICs	very high	very low	very low
FPGAs	high	low	very high

Table 2.1: Platforms comparison

System logic cells	504k
Memory	38Mb
DSP slices	1728
I/O pins	464

Table 2.2: XILINX Zynq UltraScale+ MPSoC ZCU104 features [14]

2.3.1 High-Level Synthesis languages

Hardware description languages (HDL), such as VHDL, are used to describe the design of electronic systems specifying their low-level features, allowing complete freedom to the designers. Using an HDL to develop complex systems, like this project purpose, can be discouraging, because of the time required and the difficulty of the task.

High-level synthesis languages were introduced to overcome these problems by designing the hardware using a higher level of abstraction. They are FPGA compatible, and allow designing systems using high-level specification languages, such as C, C++, SystemC, and MATLAB and inherit some of their characteristics, such as data types, structures, and loops.

This project is designed in HLS using the Vitis HLS tool. Vitis HLS allows the RTL synthesis of C++ code. Directive or pragmas are used to describe low-level characteristics, allowing different optimizations, such as pipelining and loop unrolling.

2.4 Winograd Algorithm

A standard convolution using a generic kernel $r \times s$ to compute an output $m \times n$ requires $m \times n \times r \times s$ multiplications. Winograd algorithm, introduced in 1980 by Shmuel Winograd [6], allows reducing this amount of multiplications to $(m + r - 1) \times (n + s - 1)$. This is possible to transform the convolution in an element-wise matrix multiplication ($EWMM$). To apply the algorithm the IFMaps is divided into $(m + r - 1) \times (n + s - 1)$ tiles (IFTiles) and then equation 2.5 can be used to calculate $m \times n$ OFTiles (tiles of the OFMaps).

The used notation to refer to the Winograd algorithm is $F(n \times m, r \times s)$.

The following steps summarize the procedure:

- IFTile and kernel transformation, applying two matrix multiplications to each one. After the transformations, the domain is changed from the standard to the Winograd one (WD), and inputs and weights matrices are indicated with D and W, respectively.
- EWMM between D and W, the result matrix is indicated with E.
- Inverse transformation of E, two matrix multiplications are used to restore the domain.

$$OFTile = IFTile * kernel = A^T((B \times IFTile \times B^T) \odot (G \times kernel \times G^T))A \quad (2.5)$$

A, B and G are named Winograd matrices and they change together with the kernel and output dimensions and the values of chosen *interpolation points* used to derive them with the Toom-Cook algorithm [15].

The big difference from the standard algorithm is that with Winograd multiple output pixels can be computed simultaneously. The ratio of the multiplications needed by the standard convolution and the multiplications performed using Winograd depends on the kernel and the output sizes:

$$multiplications\ reduction = \frac{m \times n \times r \times s}{(m + r - 1) \times (n + s - 1)} \quad (2.6)$$

From this relationship, it is evident that the larger the kernel and the calculated OFTile, the fewer the multiplications needed for convolution.

However, the Winograd algorithm has not only benefits but also 3 important drawbacks [16]:

- the transformations to and from the WD requires computations that can greatly increase the computational cost of a convolution
- the kernel transformation matrix has the same dimensions as the IFTile (bigger than the kernel) so it increases the on-chip memory demand

- the algorithm introduces a numerical error when combined with quantization

All these problems are more evident using larger operands' tiles.

2.5 Quantization

The use of the floating-point arithmetic (figure 2.9a) in CNNs allows for keeping numerical errors limited, avoiding negatively affecting accuracy. The complexity reached by modern networks, however, requires a large number of calculations to be performed and parameters to be stored.

Using high precision numbers in a neural network implemented in an embedded system, such as an FPGA, can be expensive in terms of memory demand, both *on-chip* and *off-chip*, and computational latency, due to the limited hardware resources.

Moving from a floating-point representation to an N-bit fixed-point quantization (figure 2.9b) can not only reduce the memory footprint but it can also allow reaching higher throughput. Working on FPGAs it is also possible to exploit the DSPs usage to parallelize integer multiplication execution, using a single DSP slice to perform two multiplications in one iteration, as explained in section (5.2).

The price to pay to quantize the parameters of a net is the introduction of a numerical error, due to the reduced set of representable values. This error can negatively affect the network accuracy.

(a) The 32-bit floating point representation uses 1 bit as sign, 8 bits as exponent and 23 bits as mantissa.

(b) The 8-bit fixed-point representation uses 1 bit as sign and 7 bits to store the value.

Figure 2.9: Comparison between the 32-bit floating point and the 8-bit fixed-point representations

2.6 Residue number system

The residue number system (RNS) is an alternative to the standard decimal system. As introduced by [10], this system can be used to represent the values involved in a Winograd convolution.

RNS$(m_0, m_1, ..., m_{n-1})$ can represent a value by n residues derived using n different pairwise coprime modules $m_0, m_1, ..., m_{n-1}$. For example, 17 in $RNS(3, 7)$

can be represented as $2, 3$. In fact:

$$17\ (mod\ 3) = 2$$

$$17\ (mod\ 7) = 3$$

It is possible to perform operations such as addition, subtraction, and multiplication in RNS and then return to the standard numerical representation using the Chinese Remainder Theorem (CRT). This is possible only if the value to be reconstructed (x) is smaller than the product (M) of all the modules used for the representation. CRT works as illustrated in the following equation.

$$x = (\sum_{i=0}^{n-1} a_i b_i N_i)(mod\ M) \tag{2.7}$$

Where:

- a_i is the representation of (x) in modulo m_i.
- N_i is the product of all the modules but m_i
- b_i is a value that satisfies the following equality: $(b_i \times N_i)(mod\ m_i) = 1$

To reconstruct x, of the previous example, from $RNS(3,7)$, where:

$$a_0 = 2$$

$$a_1 = 3$$

The values N_0 and N_1 can be easily determined as follows:

$$N_0 = m_1 = 7$$

$$N_1 = m_0 = 3$$

Correct values for b_0 and b_1 can be:

$$b_0 = 1 \rightarrow (1 \cdot 7)\ (mod\ 3) = 7\ (mod\ 3) = 1$$

$$b_1 = 5 \rightarrow (5 \cdot 3)\ (mod\ 7) = 15\ (mod\ 7) = 1$$

Applying equation 2.7:

$$x = (a_0 b_0 N_0 + a_1 b_1 N_1)\ (mod\ 21) = (2 \cdot 1 \cdot 7 + 3 \cdot 5 \cdot 3)\ (mod\ 21) =$$

$$(14 + 45)\ (mod\ 21) = 59\ (mod\ 21) = 17\ (mod\ 21)$$

Chapter 3

Related works

This chapter illustrates the works that represent the starting point of this thesis. The first paper presents an FPGA-based CNN accelerator design, the second one introduces the Winograd algorithm to the CNNs field. The last two works show how two different numerical representations can be used to reduce the numerical error introduced by the Winograd algorithm in quantized networks.

3.1 PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks

This paper [17] published in November 2016 is the baseline of this thesis work. It presents an FPGA design of a CNN accelerator based on the *OpenCL* framework. In the PipeCNN project, an FPGA board and a desktop CPU are linked together in a heterogeneous system, where the FPGA has the *device* role and the CPU is the *host*. A *C/C++ code* running on the host provides APIs that communicate with *kernels* running on the device, specified in an *OpenCL code*.

The developed architecture is composed of five deeply pipelined kernels communicating through FIFOs. Two of these kernels are in charge of reading and writing data from/to the global memory. The reader kernel stores input features and weights in local buffers. This avoids the global memory access from the computational kernels, limiting the bandwidth requirements. One of the kernels is demanded to perform the convolution, it exploits loop tiling and memory partitioning to improve throughput. Data vectorization and parallel computation units are, instead, used to implement two levels of unrolling in the structure.

3.2 Fast Algorithms for Convolutional Neural Networks

A. Lavin and S. Gray introduced, in paper [7], the Winograd algorithm to speed up the convolution process in neural networks. Both the $F(2 \times 2,3 \times 3)$ and $F(4 \times 4,3 \times 3)$ are analyzed. The first solution allows a multiplications reduction of $2.25\times$ compared to the standard convolution, while the second reaches a reduction of $4\times$. The two algorithms were tested and both reached good performance in terms of latency and accuracy. However, in these experiments, floating-point quantizations were used. Using, instead, a fixed-point low-precision quantization the Winograd approach brings a significant numerical.

3.3 Efficient Winograd Convolution Via Integer Arithmetic

The work done by Lingchuan Meng, John Brothers in 2019 [9], proposes to extend the Winograd domain from the field of rational numbers to the field of complex numbers. The purpose of this extension is to use symmetric interpolation points that allow for matrices composed of small values, thus reducing the numerical error due to the Winograd algorithm.

Looking at the $F(4 \times 4,3 \times 3)$, they observed that representing the Winograd domain using the complex field, it is composed of matrices that follow a precise pattern, exploitable to reduce optimized the computation. They take advantage of the redundancy of the conjugate complexes, halving the complex number in the matrices. When the EWMM is applied the complex numbers are multiplied recurring to the Karatsuba multiplication that makes use of 3 real multiplications. The algorithm proposed by the authors reaches a multiplications reduction of $3.13\times$.

3.4 Efficient Residue Number System Based Winograd Convolution

In [10], the Residue Number System (RNS) is used to solve the numerical error problem, introduced by the Winograd algorithm. RNS, in fact, can avoid this error, using small residues to represent higher numbers. However, it needs a Winograd domain for each number system used, so on-chip memory demand and arithmetic complexity grow linearly with the number of used modules. The survey shows that the best results in terms of arithmetic complexity reduction (compared to the

standard convolution) can be reached when the Winograd algorithm is applied to derive large output tiles (e.g 10×10).

3.5 Summary

Taking the PipeCNN work as the starting point, this project's purpose is to further optimize it. To do that, 8-bit fixed-point quantization, the insertion of a third unrolling level, and the Winograd awareness are exploited. To introduce the Winograd support in a quantized network, the complex field and the RNS are considered.

Chapter 4

Convolutional algorithms

Section 2.2.1 provides an overview of the convolutional algorithm. The six nested loops required by its computation need the execution of a huge number of multiplications, which limit the performance of a network in terms of latency. The Winograd algorithm can be used to reduce the computational cost of the convolution, but it introduces a numerical error, which can be amplified by quantization.

This chapter analyzes possible implementations of the convolution in an 8-bit quantized network, starting with the standard method, which is taken as a reference. The different solutions are evaluated in terms of computational cost and numerical error. The error is calculated through numerical simulations. The methods are evaluated by performing a 2D convolution using a 3×3 kernel (the most commonly used in CNNs).

4.1 Numerical simulations

To test the following possible solutions, which implement the Winograd algorithm, a MATLAB script was developed for each of them. These scripts evaluate the numerical errors introduced by the under-test algorithm compared to the standard quantized convolution. Therefore, each script runs two convolutions, with the same random inputs, one using the standard method and one using a Winograd possible implementation. The tests were performed running one million of these couple of convolutions.

The error is evaluated considering the maximum absolute difference and the average difference between the output features of the two convolutions. Before evaluating the differences, the two outputs are scaled to the 8-bit fixed-point representation, multiplying them by a scaling factor, evaluated as in the following equation:

$$sf = 127/max(OFMaps) \tag{4.1}$$

4.2 Standard convolution

Standard convolution requires a multiply and accumulate (MAC) operation for each filter element to derive each output pixel, so in the case of a 3×3 kernel, 9 MACs are necessary for each OFMap element.

Figure 4.1 shows the convolution of a 6×6 IFMap using a 3×3 kernel, to derive a 4×4 OFMap. It requires 144 MACs.

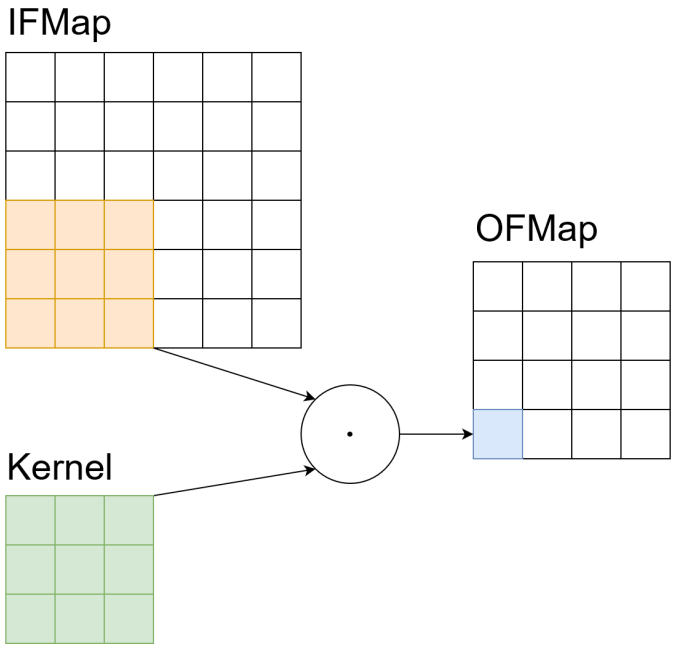


Figure 4.1: Standard convolution representation

4.3 Winograd convolution

Using the Winograd algorithm it is possible to transform an IFTile and the kernel into matrices of the same size as the IFTile(D and W). The result of an element-wise matrix (EWMM) between these matrices is the OFTile representation in the Winograd domain (E). Once E is calculated, an *inverse transformation* can be used to derive the right OFTile. Transformations to and from the Winograd domain require multiplications between matrices:

- $D = B \times IFTile \times B^T$
- $W = G \times kernel \times G^T$
- $OFTile = A^T \times E \times A$

Their contribution to the computational cost grows with the operands dimensions.

Possible implementations of the algorithm are analyzed by resorting to $F(2 \times 2, 3 \times 3)$, $F(4 \times 4, 3 \times 3)$ and $F(6 \times 6, 3 \times 3)$.

4.3.1 Winograd $F(2 \times 2, 3 \times 3)$

Winograd standard algorithm $F(2 \times 2, 3 \times 3)$ computes a 2×2 OFTile with the EWMM of two 4x4 matrices, reducing the multiplication needed by a 2.25× factor compared to the standard convolution (16 instead of 36), as shown in figure 4.2.

Looking at the A and B matrices (4.2), it is possible to notice that their values are only ones and halves, so the inputs and outputs transformations can be performed using only additions and logic operations. Also, thanks to the magnitude of the Winograd matrices elements, the numerical error introduced in a quantized network is small, during the numerical simulations the average absolute numerical error was less than 1 and the maximum was 19.

$$A_T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} B_T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} G = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

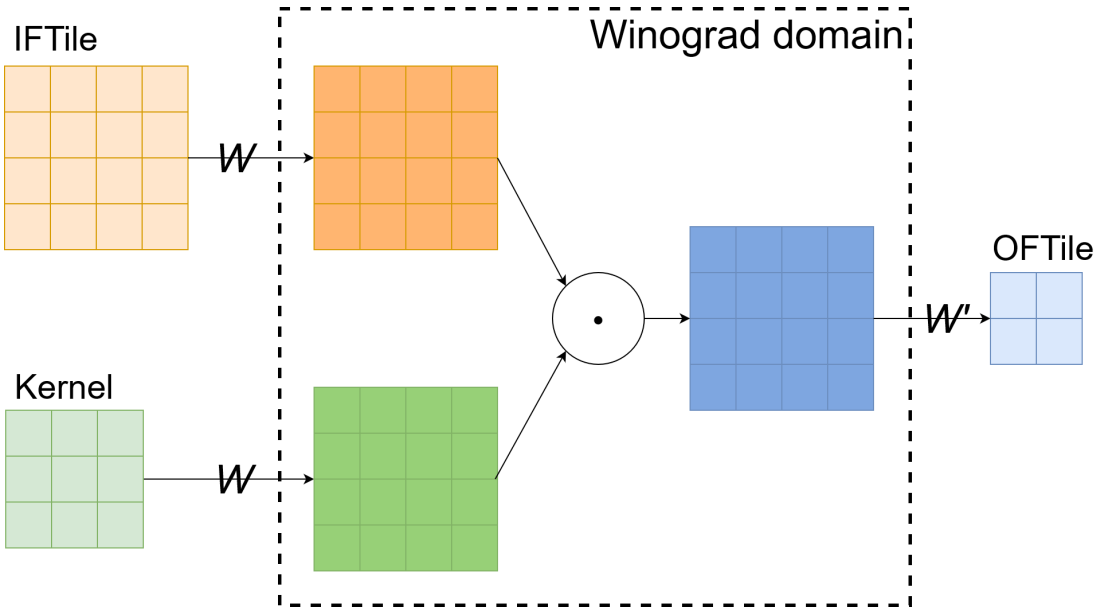


Figure 4.2: Winograd convolution $F(2 \times 2, 3 \times 3)$ representation

4.3.2 Winograd $F(4 \times 4, 3 \times 3)$

In an $F(4 \times 4, 3 \times 3)$, a 6×6 input and a 3×3 kernel are transformed resulting in 6×6 size matrices (figure 4.3). The EWMM requires only 36 multiplications to derive E against the 144 required by standard convolution to calculate a 4×4 output (4× reduction). Analyzing the A and B Winograd matrices (4.3), it is possible to notice that they are composed of powers of 2 or a small integer number (-5), so the transformations can be performed by additions and logic operations.

This algorithm works well in a single or double precision net, but when it is quantized to 8-bit, as in the analyzed case, it introduces a high numerical error. This error is due to the wide numerical range between A, B, and G values [16].

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

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ \frac{-1}{6} & \frac{-1}{6} & \frac{-1}{6} \\ \frac{-1}{6} & \frac{1}{6} & \frac{-1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & \frac{-1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix} \tag{4.3}$$

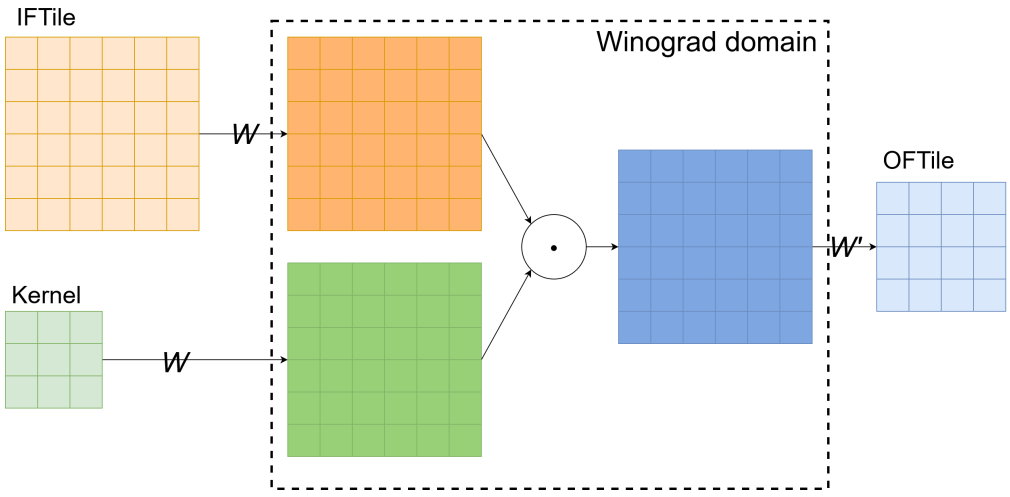


Figure 4.3: Winograd convolution $F(4 \times 4, 3 \times 3)$ representation

4.3.3 Winograd $F(6 \times 6, 3 \times 3)$

Solving formula 2.6 for an $F(6 \times 6, 3 \times 3)$, the multiplications reduction is equal to 5.06×, which is the highest reduction among the analyzed solution. Figure 4.4 shows the procedure: inputs and kernels are transformed in 8×8 matrices to be multiplied in an EWMM. The number of multiplications needed to compute a 6×6 output is 64 instead 324 as the standard convolution.

However, looking at the Winograd matrices 4.4 two problems are evident:

- the complexity of the matrices elements considerably increases the computational cost of the transformations, introducing the need for multiplications
- the numerical interval is very high (the maximum ratio between two values is equal to 2880) and so the numerical error cannot be avoided

$$
B^T = \begin{bmatrix}
1 & 0 & -21/4 & 0 & 21/4 & 0 & -1 & 0 \\
0 & 1 & 1 & -17/4 & -17/4 & 1 & 1 & 0 \\
0 & -1 & 1 & 17/4 & -17/4 & -1 & 1 & 0 \\
0 & 1/2 & 1/4 & -5/2 & -5/4 & 2 & 1 & 0 \\
0 & -1/2 & 1/4 & 5/2 & -5/4 & -2 & 1 & 0 \\
0 & 2 & 4 & -5/2 & -5 & 1/2 & 1 & 0 \\
0 & -2 & 4 & 5/2 & -5 & -1/2 & 1 & 0 \\
0 & -1 & 0 & 21/4 & 0 & -21/4 & 0 & 1
\end{bmatrix}
$$

$$
G = \begin{bmatrix}
1 & 0 & 0 \\
\frac{-2}{9} & \frac{-2}{9} & \frac{-2}{9} \\
\frac{-2}{9} & \frac{2}{9} & \frac{-2}{9} \\
\frac{1}{90} & \frac{1}{45} & \frac{2}{45} \\
\frac{1}{90} & \frac{-1}{45} & \frac{2}{45} \\
\frac{32}{45} & \frac{16}{45} & \frac{8}{45} \\
\frac{32}{45} & \frac{-16}{45} & \frac{8}{45} \\
0 & 0 & 1
\end{bmatrix}
\quad
A^T = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & -1 & 2 & -2 & 1/2 & -1/2 & 0 \\
0 & 1 & 1 & 4 & 4 & 1/4 & 1/4 & 0 \\
0 & 1 & -1 & 8 & -8 & 1/8 & -1/8 & 0 \\
0 & 1 & 1 & 16 & 16 & 1/16 & 1/16 & 0 \\
0 & 1 & -1 & 32 & -32 & 1/32 & -1/32 & 1
\end{bmatrix}
\tag{4.4}
$$

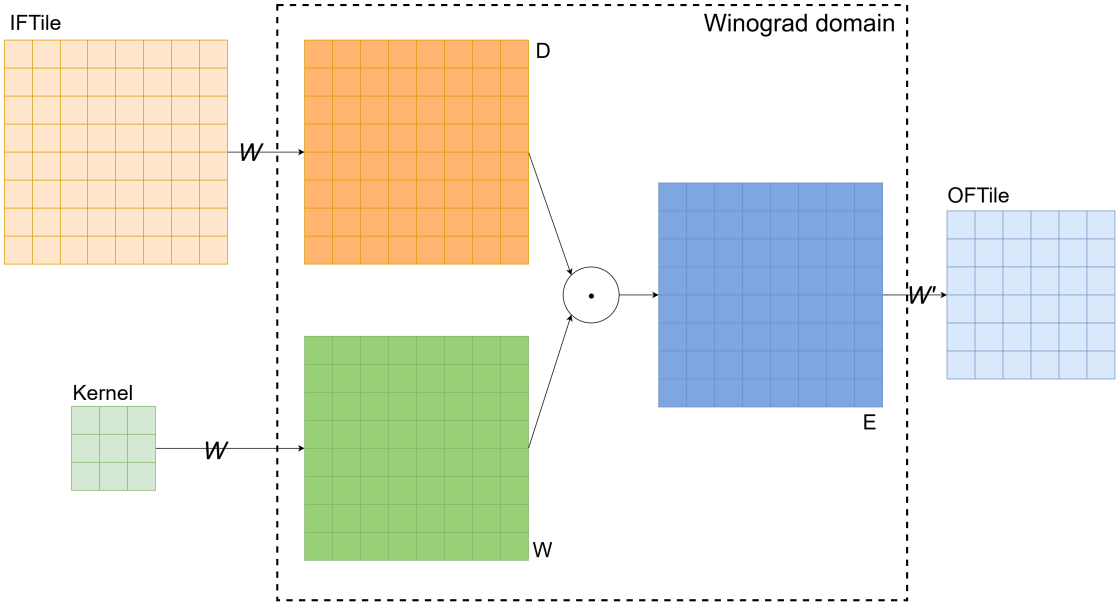


Figure 4.4: Winograd convolution $F(6 \times 6, 3 \times 3)$ representation

4.4 Complex Winograd convolution

Complex Winograd convolution was introduced to solve the problem of transformation matrices elements. The complex field can provide new interpolation points that can be exploited to derive simpler matrices than in the standard Winograd. The price to pay is the representation of complex numbers and their multiplications.

Complex $F(2 \times 2, 3 \times 3)$ is not needed because standard Winograd already has good matrices. In the case of $F(6 \times 6, 3 \times 3)$, it is not possible to find interpolation points that significantly reduce the numerical error, in every possible configuration the matrix G presents fractional numbers with a very high denominator. The only algorithm, among those analyzed, that can take advantage of the complex representation, using 8-bit data, is $F(4 \times 4, 3 \times 3)$.

4.4.1 Complex Winograd $F(4 \times 4, 3 \times 3)$

To derive the Winograd matrices of an $F(4 \times 4, 3 \times 3)$ it is possible to use the interpolation set $[0, 1, -1, i, -i]$. The symmetry in the interpolation points contributes to reducing the magnitude of the elements in the matrices and this leads to a significant reduction of the numerical error.

Another positive aspect of matrices values reduced magnitude, is that the

Winograd transformations are simpler to be implemented using logical elements.

$$B^T = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & -1 & 1 & -1 & 1 & 0 \\ 0 & -i & -1 & i & 1 & 0 \\ 0 & i & -1 & -i & 1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ 0 & 0 & 1 \end{bmatrix} A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & i & -i & 0 \\ 0 & 1 & 1 & -1 & -1 & 0 \\ 0 & 1 & -1 & -i & i & 1 \end{bmatrix} \tag{4.5}$$

Being the matrices in the Winograd domain composed of complex values, both the real and imaginary parts of these values must be represented. From the analysis of the matrices W and D, it is possible to see that 20 out of 36 elements are complex and they are pairs of conjugate complexes (figure 4.5).

$R_{0,0}$	$R_{1,0}$	$R_{2,0}$	$C_{3,0}$	$\overline{C_{3,0}}$	$R_{5,0}$
$R_{0,1}$	$R_{1,1}$	$R_{2,1}$	$C_{3,1}$	$\overline{C_{3,1}}$	$R_{5,1}$
$R_{0,2}$	$R_{1,2}$	$R_{2,2}$	$C_{3,2}$	$\overline{C_{3,2}}$	$R_{5,2}$
$C_{0,3}$	$C_{1,3}$	$C_{2,3}$	$C_{3,3}$	$C_{4,3}$	$C_{5,3}$
$\overline{C_{0,3}}$	$\overline{C_{1,3}}$	$\overline{C_{2,3}}$	$\overline{C_{3,3}}$	$\overline{C_{4,3}}$	$\overline{C_{5,3}}$
$R_{0,5}$	$R_{1,5}$	$R_{2,5}$	$C_{3,5}$	$\overline{C_{3,5}}$	$R_{5,5}$

Figure 4.5: Matrices pattern in complex Winograd domain in $F(4 \times 4,3 \times 3)$

Taking advantage of the redundancy of the conjugate complexes, it is possible to consider only half of the 20 that are present in the matrices. There are, therefore, only 10 complex multiplications to perform in the EWMM, and using Karatsuba multiplication (equation 4.6), each of them can be replaced by 3 real multiplications, so the total number of multiplications in an $F(4 \times 4,3 \times 3)$ is 46 instead of the 144 of a standard convolution, leading to a multiplication reduction of 3.13×.

$$\begin{gathered} X = x_0 + x_1 i,\ Y = y_0 + y_1 i \\ XY = (x_0 y_0 - x_1 y_1) + ((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1) i \end{gathered} \tag{4.6}$$

Figure 4.6 shows how the complex $F(4 \times 4{,}3 \times 3)$ is applied. The complex values of the matrices in WD are darker than the other values.

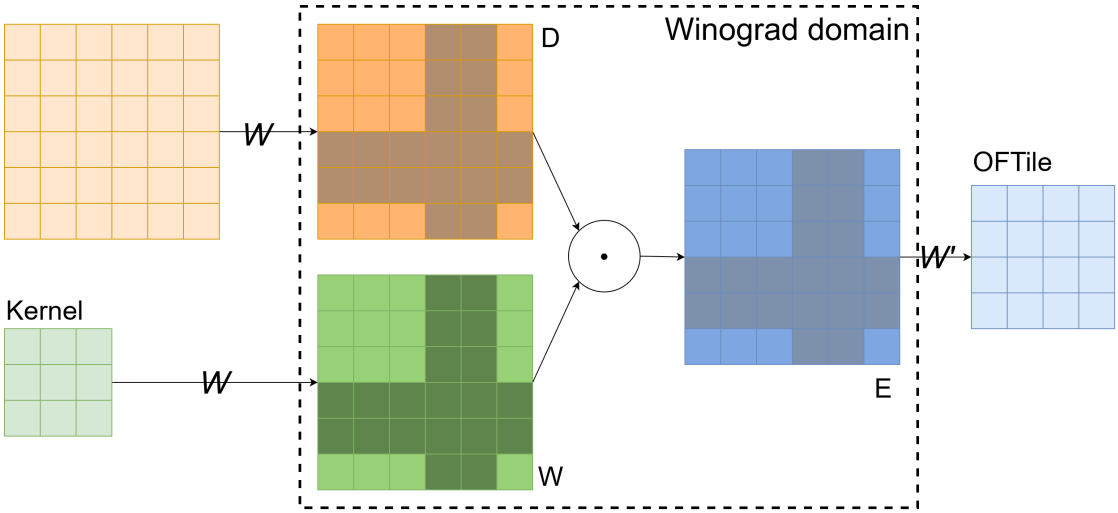


Figure 4.6: Complex Winograd convolution $F(4 \times 4, 3 \times 3)$ representation

4.5 RNS Winograd convolution

RNS Winograd is the only way, among the analyzed solutions, to provide the same results as the standard convolution in a quantized network. To avoid it, the modules chosen to derive the residues must follow the rules expressed in section 2.6:

- modules must be pairwise coprime
- the numerical range width representable in RNS is equal to the product of the modules

To comply with the first rule in an 8-bit quantized network it is possible to search between prime numbers less than 2^8. Choosing the two highest possible values $(251, 241)$ the representable range is $[-30245; 30245]$. This interval doesn't allow the correct representation of the convolution. Adding 239 to the module set, on the other hand, makes it possible to include all possible values reached by the operation. This means that at least 3 residue systems are needed to represent the convolution.

4.5.1 RNS$(239, 241, 251)$ Winograd $F(4 \times 4, 3 \times 3)$

To apply the RNS Winograd method (Figure 4.7), the residues of the transformation matrices are used, instead of the standard ones. In $F(4\times4, 3\times3)$, since the matrices A and B are composed only of integers, smaller than the chosen modules, they are the same among the three systems, thus, the D matrix can be shared. Matrix G, on the other hand, is also composed of fractional numbers, which require three different G matrices and so 3 different representations of W matrices. It is therefore necessary to triple the number of EWMMs to compute three different E matrices, from which to derive three output matrices, on which to apply CRT to derive the OFTile. Thus, without considering the CRT operation, the multiplications reductions drop to 1.33×.

As for Winograd transformations, they can be carried out by logical elements, because the same considerations on the A and B matrices of the standard $F(4 \times 4, 3 \times 3)$ are valid.

The main con of this solution is the CRT operation to be performed at the end of the operation, which requires 6 multiplications for each output pixel.

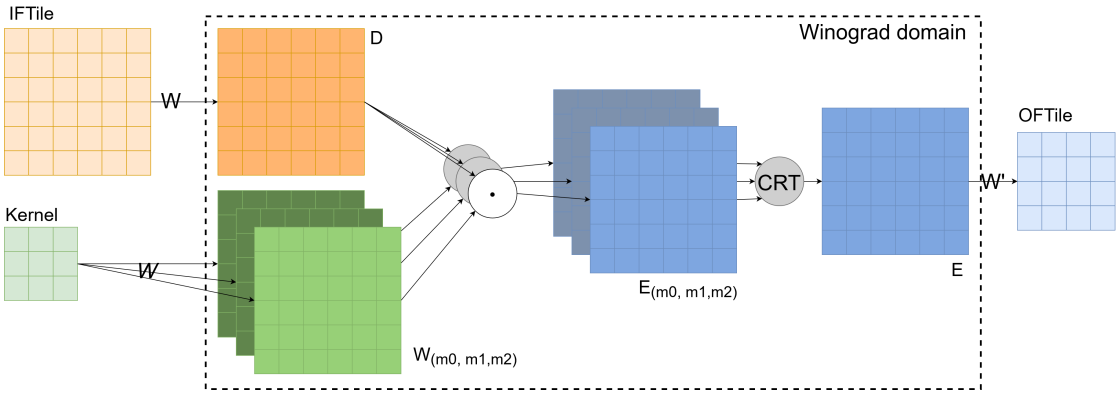


Figure 4.7: RNS(m_0, m_1, m_2) Winograd convolution $F(4 \times 4, 3 \times 3)$ representation

4.5.2 RNS$(239, 241, 251)$ Winograd $F(6 \times 6, 3 \times 3)$

In the case of $F(6 \times 6, 3 \times 3)$, the same observations made on matrices A and B are not valid. They are composed of fractional numbers which require different representations in the 3 number systems. This greatly increases the number of multiplications needed, canceling the advantage derived from the Winograd algorithm.

4.6 Algorithms comparison

Table 4.1 represents a comparison between the analyzed methods, summarizing the previous sections. The three parameters analyzed are complexity reduction (CR), maximum absolute error (MAX err), and average absolute error (AVG err). In the CR column only the multiplications used in the EWMMs are considered compared to the quantized standard convolution. Also, the numerical errors are referred to the standard algorithm and their values are derived as described in section 4.1.

Standard Winograd solutions $F(4 \times 4, 3 \times 3)$ and $F(6 \times 6, 3 \times 3)$ are the bests in terms of the number of multiplications, but they are unreliable.

The RNS Winograd $F(4 \times 4, 3 \times 3)$ is the only solution able to avoid the numerical error, but, it requires the CRT operation that introduces multiplications of each output pixel and has a higher memory footprint, due to the tripled WD representation. For these reasons, the RNS Winograd doesn't represent the best solution among those analyzed. However, it can be of interest to analyze the RNS benefits using higher operands dimensions or higher precision quantization in the Winograd domain, because of the higher multiplications reduction in the first case, and the possibility to use higher modules and so only 2 systems in the second case.

Complex Winograd $F(4 \times 4, 3 \times 3)$ represents a trade-off between arithmetic

complexity and numerical error. Looking at the computational cost this solution is also better than $F(2 \times 2, 3 \times 3)$ because it allows a higher reduction (3.13× against 2.25×).

Winograd algorithm	muls reduction	MAX err	AVG err
$F(2 \times 2, 3 \times 3)$	2.25×	19	0.76
$F(4 \times 4, 3 \times 3)$	4×	256	24.7
$F(6 \times 6, 3 \times 3)$	5.06×	256	38.16
Complex $F(4 \times 4, 3 \times 3)$	3.13×	18	1.53
RNS$(239, 241, 251)$ $F(4 \times 4, 3 \times 3)$	1.33×	0	0

Table 4.1: Winograd algorithms comparison

Chapter 5

Methodologies

5.1 Winograd awareness

In chapter 4 analysis, Complex Winograd $F(4 \times 4, 3 \times 3)$ represents the best trade-off between numerical error and multiplications number. This section shows the changes needed to make the PipeCNN [17] convolutional kernel aware of this Winograd algorithm and the techniques used to optimize it.

5.1.1 IFMap tile reading and transformation

To apply a generic $F(m \times n, r \times s)$ Winograd algorithm to an input of generic size each of its channels must be divided in $(m + r - 1) \times (n + s - 1)$ tiles, with $r - 1$ and $s - 1$ overlapped elements in the corresponding dimensions [7]. This operation can be performed offline, thus the inputs can be read as vectorized tiles and the algorithm can be applied, starting from their transformation. Complex Winograd $F(4 \times 4, 3 \times 3)$ needs (6×6) tiles as input. The transformation of these tiles results in 36 different elements.

5.1.2 Offline weights transformation

The chosen Winograd algorithm provides a fractional G matrix. This matrix is involved in the kernel transformation. Performing this operation using integer arithmetic can introduce a significant loss of information. An offline kernel transformation computed using a single-precision floating-point makes it possible to directly read the 8-bit quantized transformed weights from the off-chip memory, reducing the numerical error, but also avoiding the weights transformation computational costs.

Once computed, the W matrix can also be scaled using a scaling factor to represent weights with the whole admitted range by the quantization, reducing the

numerical error even more. In addition, this operation is performed offline.

5.1.3 Hardware sharing between standard convolution and EWMM

A convolutional kernel aware of Complex Winograd $F(4 \times 4, 3 \times 3)$ also needs to support standard convolution to keep the compatibility with different convolution parameters, such as stride or kernel dimensions.

Both algorithms make a massive usage of multiplications. The best way to efficiently implement them in FPGA is using dedicated digital signal processors (DSPs). However, the number of DSPs in an FPGA is limited, so to take full advantage of them, they can be shared by the two convolution methods.

5.1.4 Outputs transformation

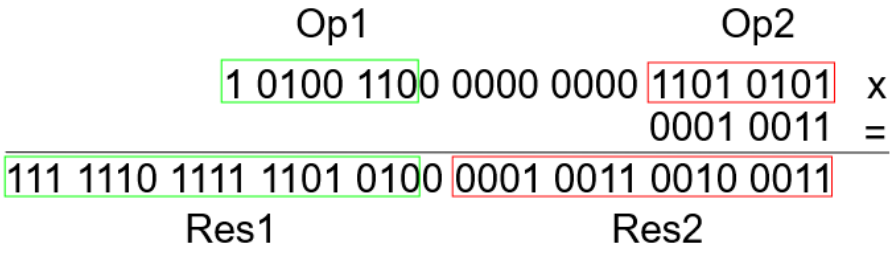
The last step to make a convolutional kernel aware of Complex Winograd $F(4 \times 4, 3 \times 3)$ is to implement its output transformation. This is the most expensive part in terms of used logic because of the number of addictions and shifts required, but also because of the width of the processed values. They are 32-bit wide since they store the accumulated products of the EWMMs.

5.2 Two multiplication in one DSP

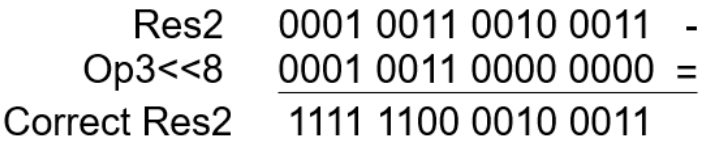
As explained in section 5.1.3, multiplications execution is demanded to DSPs. A possible optimization to further exploit the available DSPs is to simultaneously perform two multiplications in the same DSP [18]. In these two multiplications, the two multiplicands (i.e. the first operands) are different, but the multiplicator (i.e. the second operand) is the same. In a multiplication between n-bit wide operands it is possible to apply this method if the following two conditions are met:

- the output value of the DSP must be at least 4n bits
- one of the two input values must be mapped with the two multiplicands separated by at least n bits

For example, it is possible to map two multiplications, both signed or unsigned, of two 8-bit operands using a single DSP that performs a multiplication between a 25-bit value and another value 8 bit wide. The 25-bit input is the one with the two multiplicands (separated by 9 bits).



(a) Multiplication between the 25-bit and 8-bit operands



(b) A correction factor is applied to derive the final Res2

Figure 5.1: Example of two multiplications simultaneously performed

Figure 5.1a shows the multiplicator (23) that simultaneously multiplies the multiplicands (-90 and -43). It can be seen that the second result is incorrect: the returned value is 10787 instead of -989. The reason behind this is that in the case of a negative first operand, multiplication alone is not enough. It is necessary to subtract to the second result the multiplicator (23) shifted by n positions to make the correct result succeed. This applies only to the multiplication performed in the least significant bits, as the multiplier in the DSP correctly extends the sign in the most significant bits.

The XILINX ZCU104 DPSs (DSP48) have multipliers with 45-bit wide outputs. In the design proposed in chapter 6 multiplicands values are 9-bit and 12-bit wide, thus, their multiplication can be mapped as shown in figure 5.2.

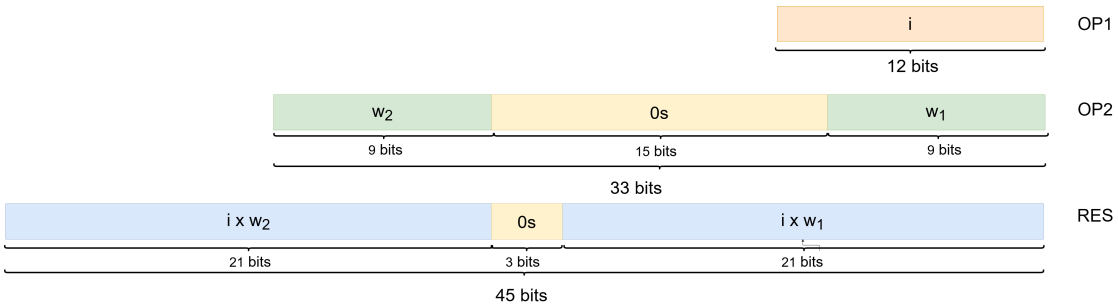


Figure 5.2: DSP registers mapping

5.3 Loop unrolling

Unrolling a loop with a P factor means reducing the number of its iteration P times, parallelizing its loop body execution. Being the convolutional algorithm a huge loop, its execution can be greatly improved by the loop unrolling usage.

This project exploits three levels of unrolling (figure 5.3), corresponding to three different dimensions of the convolution:

- P_{kx}: kernel x dimension
- P_{if}: input channel dimension
- P_{of}: output channel dimension

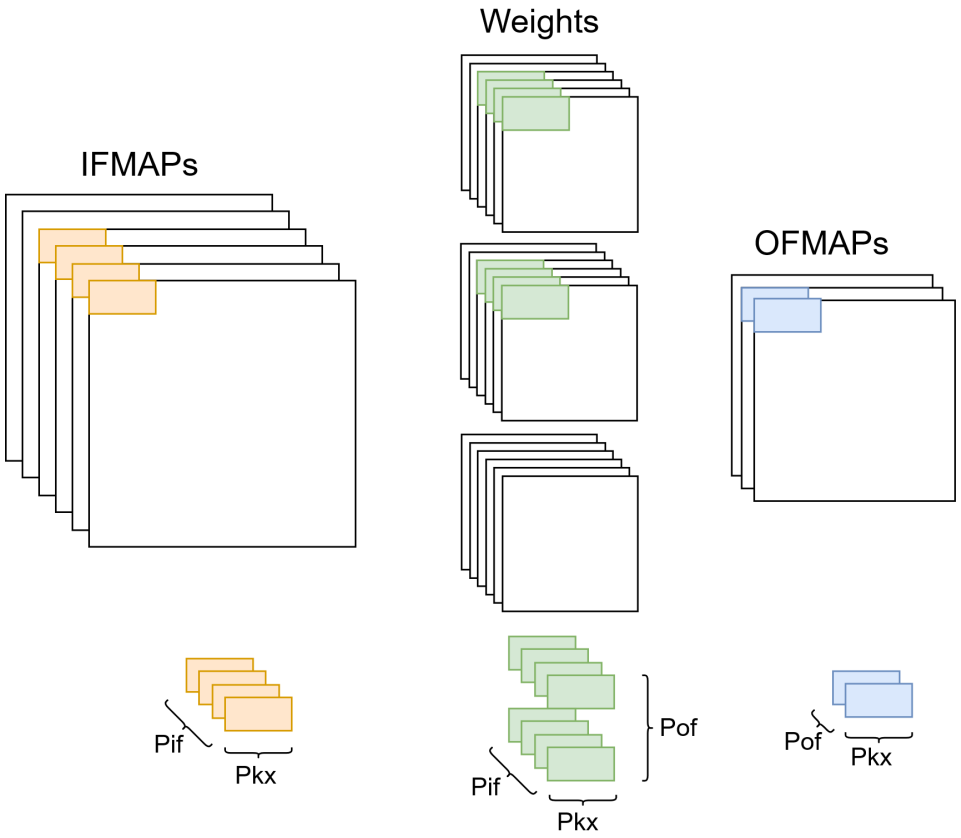


Figure 5.3: 3D unrolling of a convolution

These unrolling factors reduce the convolutional loop iterations by $P_{kx} \times P_{if} \times P_{of}$. The price to pay, however, is a higher hardware demand that grows linearly with the unrolling levels. For example, looking at the DSPs usage, mapping

two multiplications in each of them, $\frac{P_{kx} \times P_{if} \times P_{of}}{2}$ DSPs are required. Also, on-chip memory demand and logical element usage are negatively affected by the unrolling. The maximum level of possible unrolling is so set by the available hardware. Unrolling the convolutional loop (algorithm 2) using the three introduced parameters it is possible to derive the following unrolled algorithm:

Algorithm 2 Unrolled convolutional loop

```
for oy = 0; oy < N_oy; oy++ do
    for ox = 0; ox < N_ox; ox++ do
        for ky = 0; ky < N_ky; ky++ do
            for of = 0; of < N_of/P_of; of++ do
                for if = 0; if < N_if/P_if; if++ do
                    for kx = 0; kx < N_kx/P_kx; kx++ do
                        OFMaps(ox, oy, of)+=
                        Weights(kx, ky, if, of) ×IFMaps(s × ox + kx, s × oy + ky, if)
                        ...

                        OFMaps(ox, oy, of + Pof)+ =
                        Weights(kx + P_kx, ky, if + Pif, of + Pof) ×
                        IFMaps(s × ox + kx + P_kx, s × oy + ky, if + Pif)
                        end
                    end
                end
            end
        end
    end
```

5.4 Data vectorization

To make the convolutional loop unrolling possible, as described in the previous section, $P_{kx} \times P_{if} \times P_{of}$ weights and $P_{kx} \times P_{if}$ inputs must be available at the same time. To do that, it is possible to handle inputs and weights values as a set of vectors. These vectors can be composed of P_{if} overlapped values in the input channel dimension. To correctly feed the convolutional loop, inputs and weights must be grouped as P_{kx} vectors and P_{of} sets of P_{kx} vectors, respectively. When reading values as vectors it can be possible that operands sizes are not multiple of the unrolling parameters; in this case, it is useful to map with zeros the read vectors in the exceeding positions. For example, being weights size $3 \times 3 \times 64 \times 64$, if all the unrolling parameters are equal to 2, each kernel can be read as if it was a 4×3 matrix (figure 5.4). Thus, more data than needed are processed, wasting

multiplications and resources. This, however, allows a parallel implementation that speeds up the execution of the convolutional loop.

Figure 5.4: 3×3 kernel 0 mapping using $P_{kx} = 2$

Chapter 6

IP design

The following sections describe the designed IP. The proposed architecture can perform the convolution using both the standard method and Complex Winograd $F(4 \times 4, 3 \times 3)$ for an 8-bit fixed-point quantized neural network and it can be adapted to different unrolling levels, honoring the FPGA's flexibility.

6.1 Design flow

The structure of the developed system is composed of two main elements, which cover the two different roles of *host* and *device*:

- a $C++$ code is run by a testbench, on a CPU (host), which has the task of sending the input stimuli to the device, and of receiving the outputs, to be evaluated

- an HLS code, compiled and synthesized to run on the XILINX FPGA (device), which describes parallel compute units using kernel functions

Figure 6.1 illustrates the two development processes for host and device code, that use separate compilers. The communication between host and device is demanded to streaming data structures (streams), that are implemented as external ports of the device.

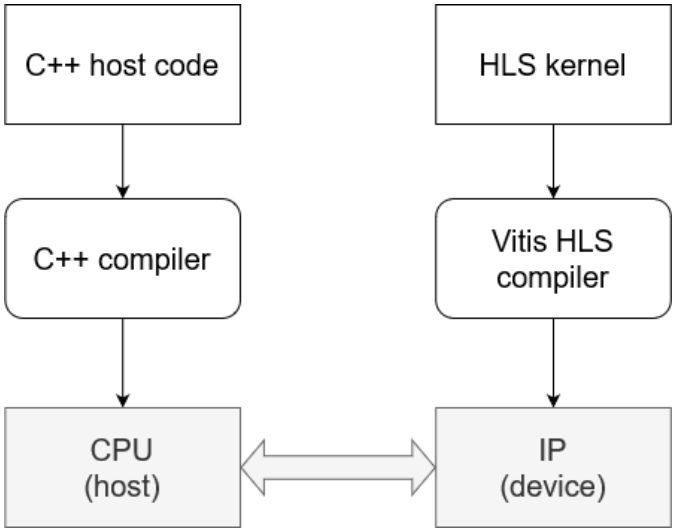


Figure 6.1: Vitis HLS based design flow

6.2 Kernels structure

The IP structure is organized in four different pipelined kernels, communicating through streams implemented as internal FIFOs. The four kernels are:

- Input transformation
- Weights reading
- Processing kernel
- Output transformation

In addition, each kernel exploits data vectorization and loop unrolling, to improve the pipeline usage and therefore the computational throughput.

Since standard convolution doesn't require transformations, the two types of convolution follow two different datapaths. Standard convolution datapath is shorter and it is composed only of the testbench and the processing kernel (represented in blue in figure 6.2); the Winograd convolution datapath, instead, passes through all the four kernels (represented in yellow).

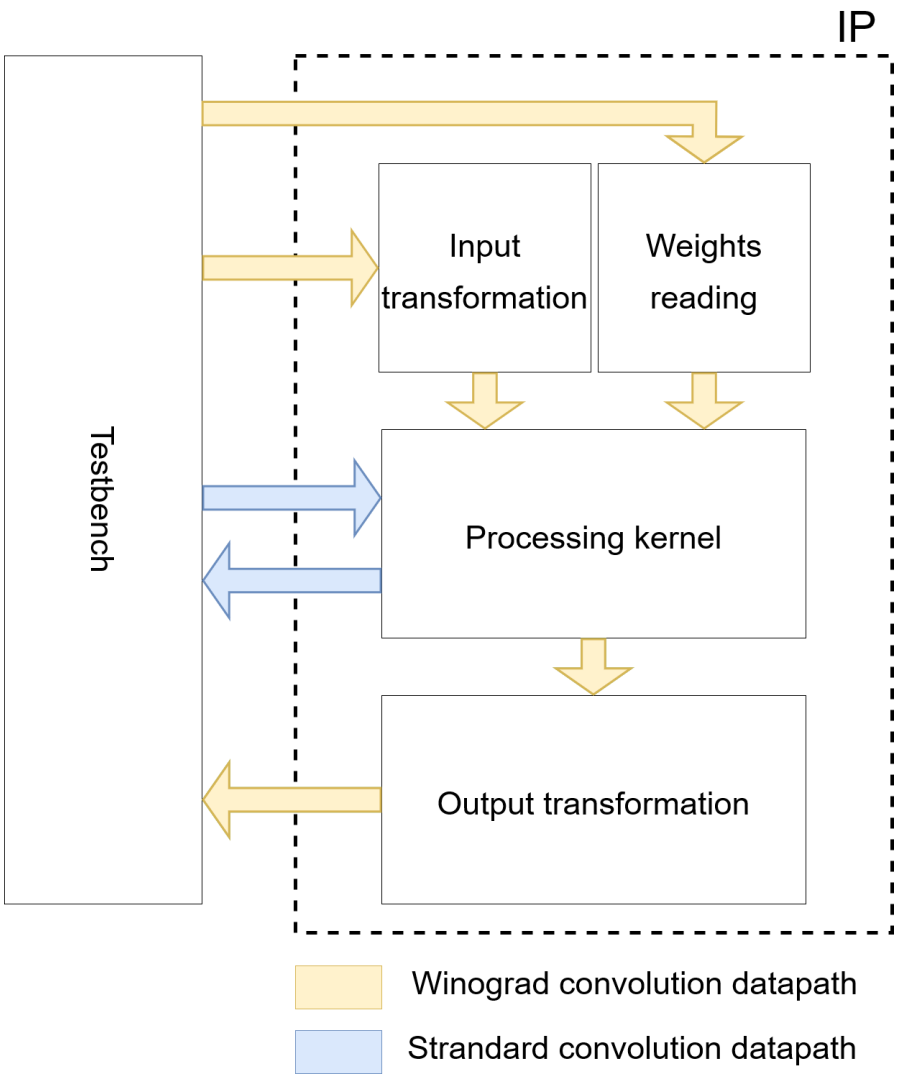


Figure 6.2: IP kernels structure

6.2.1 Input transformation (IT)

The input transformation kernel (figure 6.3) is used to read the input data from the testbench and to feed the processing kernel with the transformed values. It reads streams of $P_{if} \times P_{kx}$ 8-bit data from an external port, communicating with the testbench. These data are collected in P_{if} arrays of 36 elements, which represent the IFMaps 6×6 tiles, inputs of the Winograd algorithm.

Once filled, the set of the array is simultaneously transformed into a P_{if} set of 46 elements vectors, recurring only to additions.

The 46 elements of the output vectors are divided as follows:

- 26 values representing the real part of the D matrix
- 10 values representing the imaginary part of the D matrix
- 10 values representing the sum of the real and imaginary parts of each couple of complex conjugates

Only 10 of the 20 complex numbers are represented. This is because the redundancy of the complex numbers has been exploited (section 4.4.1). The last 10 values are used as operands of the third multiplication of the Karatsuba algorithm (equation 4.6).

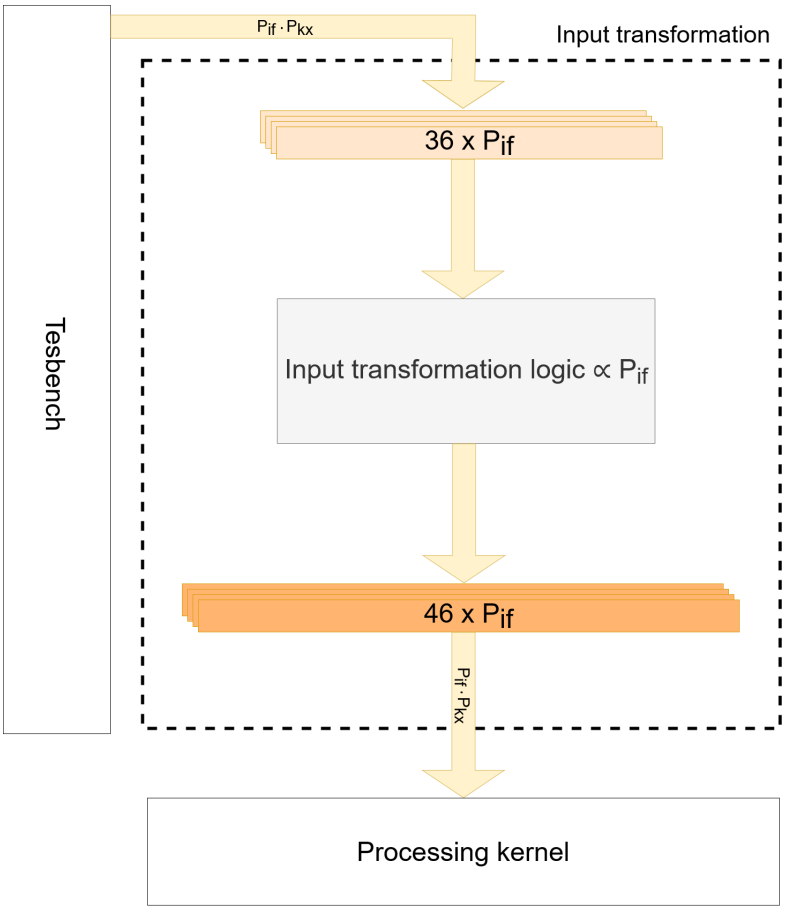


Figure 6.3: Input transformation kernel

Since the additions, used in the transformation, sums together up to 12 IFMaps elements, the output data requires 4 bits more than the inputs, so the output vectors' data-width is about 12 bits.

In each iteration, $P_{if} \times P_{kx}$ outputs are written in the FIFO that communicates with the processing kernel.

The kernel function is implemented as follows:

Algorithm 3 Input transformation loop

```
for of = 0; of < ⌈N_of / P_of⌉; of++ do
    for xy = 0; xy < ⌈N_if / P_if⌉; xy++ do
        for if = 0; if < (N_ox × N_oy) / 16; if++ do
            for r = 0; r < ⌈36 / P_kx⌉; r++ do
                read P_if × P_kx element from TB
            end
            transformation of P_if arrays
            for s = 0; s < ⌈46 / P_kx⌉; s++ do
                send P_if × P_kx element to PK
            end
        end
    end
end
```

At the and of the execution, $\lceil \frac{N_{of}}{P_{of}} \rceil \times \lceil \frac{N_{if}}{P_{if}} \rceil \times \frac{N_{ox} \times N_{oy}}{16}$ iterations of the following loop are executed:

- $\lceil \frac{36}{P_{kx}} \rceil$ stream reading (of $P_{if} \times P_{kx}$ value)
- P_{if} arrays transformation
- $\lceil \frac{46}{P_{kx}} \rceil$ stream writing (of $P_{if} \times P_{kx}$ value)

It is evident how by increasing the unrolling levels, the number of iterations decreases. However, the higher the P_{if} is, the higher is the number of parallel processed matrix transformations are, and so the logical elements usage increases.

6.2.2 Weights reading (WR)

Weights transformation is not needed because it is performed off-chip (section 5.1.2). However, only 36 values of the 46 needed by the processing kernel are provided from the testbench. The missing 10 values are the operands of the third multiplication in the Karatsuba algorithm (4.6). 10 additions are used to evaluate them.

The weights reading kernel is in charge of reading the input weights, performing the 10 sums, and sending the 46 values to the processing kernel.

$P_{of} \times P_{if}$ vectors of 36 values are processed in parallel, resulting in $P_{of} \times P_{if}$ vectors of 46 values divided as in the input transformation case.

In one iteration the kernel reads $P_{of} \times P_{if} \times P_{kx}$ 8-bit weights from the external port, communicating with the testbench, and writes the same amount of output weights in the FIFO communicating with the processing kernel. The output weights are 9 bits wide because of the sums.

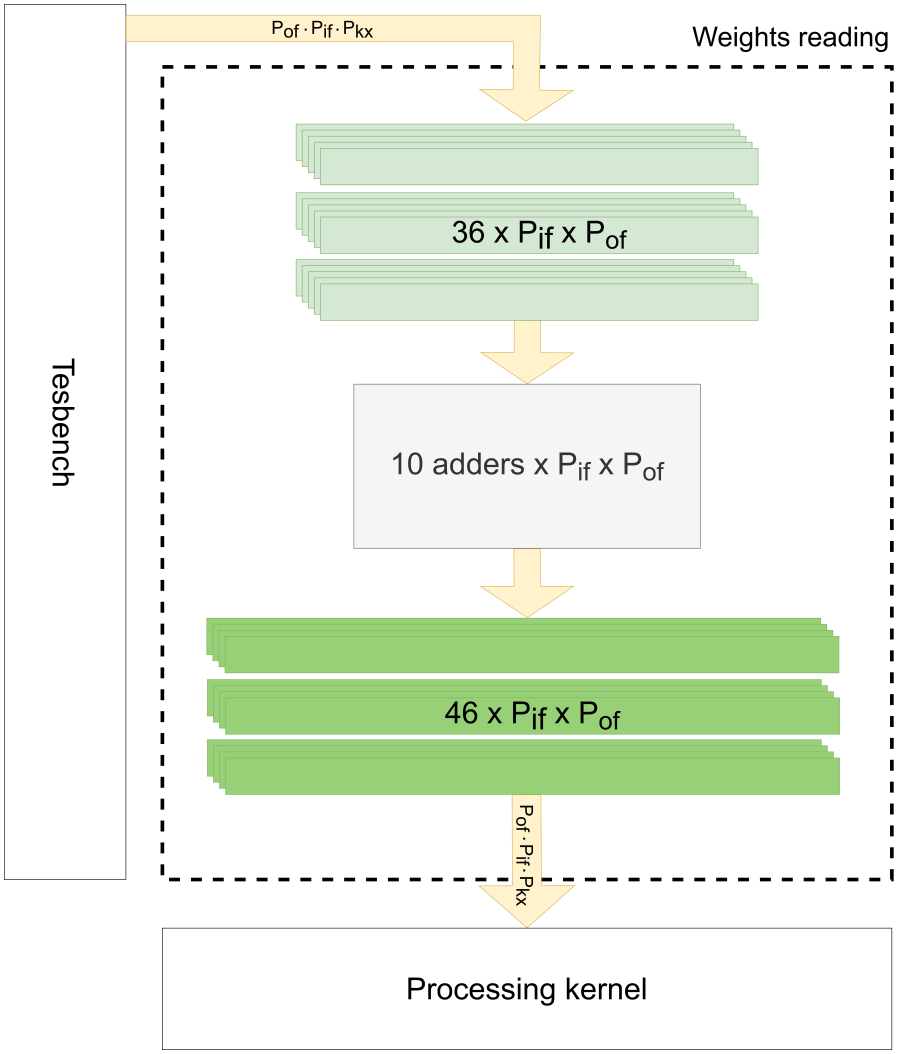


Figure 6.4: Weights reading kernel

The weights reading kernel function is described by the pseudo-code in algorithm 4. The number of total executions of its tasks is $\lceil \frac{N_{of}}{P_{of}} \rceil \times \lceil \frac{N_{if}}{P_{if}} \rceil$, therefore, is $\frac{N_{ox} \times N_{oy}}{16}$ times less than the input transformations executions. This is because of the weights reuse done by the processing kernel. In addition, in this case, increasing the unrolling levels the number of iterations decreases and the logical elements demand increases.

Algorithm 4 Weights reading loop

```
for of = 0; of < ⌈N_of / P_of⌉; of++ do
    for xy = 0; xy < ⌈N_if / P_if⌉; xy++ do
        for r = 0; r < ⌈36 / P_kx⌉; r++ do
            read P_of × P_if × P_kx element from TB
        end
        10 × P_of × P_if sums
        for s = 0; s < ⌈46 / P_kx⌉; s++ do
            send P_of × P_if × P_kx element to PK
        end
    end
end
```

6.2.3 Processing kernel (PK)

The Processing kernel is the core of the IP. It is demanded to perform the multiplications required by both the supported algorithms. It is the only kernel shared among the two datapaths.

Each weight is read once and then it is stored in a local buffer. Weights are read $P_{of} \times P_{if} \times P_{kx}$ at a time, selecting from the streams connected to the testbench or the weights reading kernel. Read weights are stored in a local buffer, to be reused with different inputs, saving off-chip memory accesses.

In each iteration the kernel reads $P_{if} \times P_{kx}$ inputs data, selecting between the two streams connected to the input transformation kernel or the testbench.

The read values feed the basic element of this kernel, a systolic array [19] of processing elements (PEs) [20] (figure 6.5). The dimensions of the PEs array are equal to the unrolling dimensions: they are grouped in P_{of} matrices of $P_{if} \times P_{kx}$ elements (figure 6.7). The inputs values are shared between the different matrices, weights are not shared.

Each PE performs a multiplication between an input and a weight and accumulates its result with the result of the previous PE in the P_{if} dimension.

Couples of PEs, belonging to two neighboring PE matrices, share the same DSP to perform their multiplications. They multiply the same input with two different weights, so it is possible to map the DSPs as illustrated in section 5.2. It is, therefore, possible to perform $P_{of} \times P_{if} \times P_{kx}$ multiplications in one iteration using $\frac{P_{of} \times P_{if} \times P_{kx}}{2}$ DSPs.

The outputs of the systolic array follow two different paths based on the applied algorithm:

- In the case of standard convolution, the values are also summed in the P_{kx}

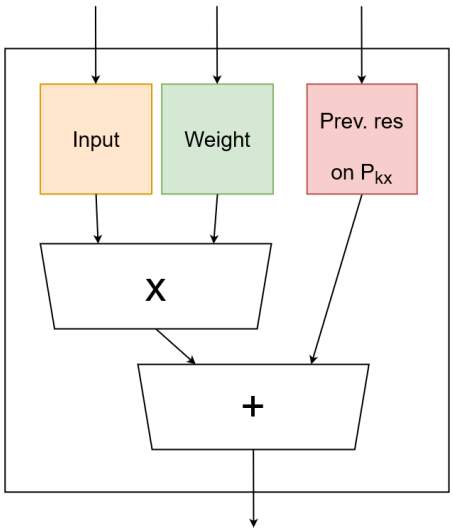


Figure 6.5: Processing elements structure

dimension, using P_{of} adder trees, and then the sums are sent to the testbench P_{of} each iteration

- In the case of a Winograd convolution, values are sent directly to the output transformation kernel through the correspondent FIFO, without an accumulation in the P_{kx} dimension, so preserving the E matrices values. The FIFO is filled with $P_{kx} \times P_{of}$ values each iteration

In both cases, the output values are 32-bit wide.

The code in algorithm 5 shows how the two convolutions are implemented in the same kernel. A boolean variable set by the testbench is used to select between the two different paths. The iterations number of two loops depends on the applied algorithm. One of these two loops spans the output xy dimensions, it differs because the $F(4\times4,3\times3)$ produces 16 outputs at times while the standard convolution only one. The other loop is the one that applies the multiplications and accumulations, it differs because in the case of the Winograd algorithm the operations are performed on 46 elements vectors, while the standard convolution is applied on vectors of the same dimensions of the unrolling level. The derivation of the two iterations parameter is performed by the testbench and it is shown in blue in the code.

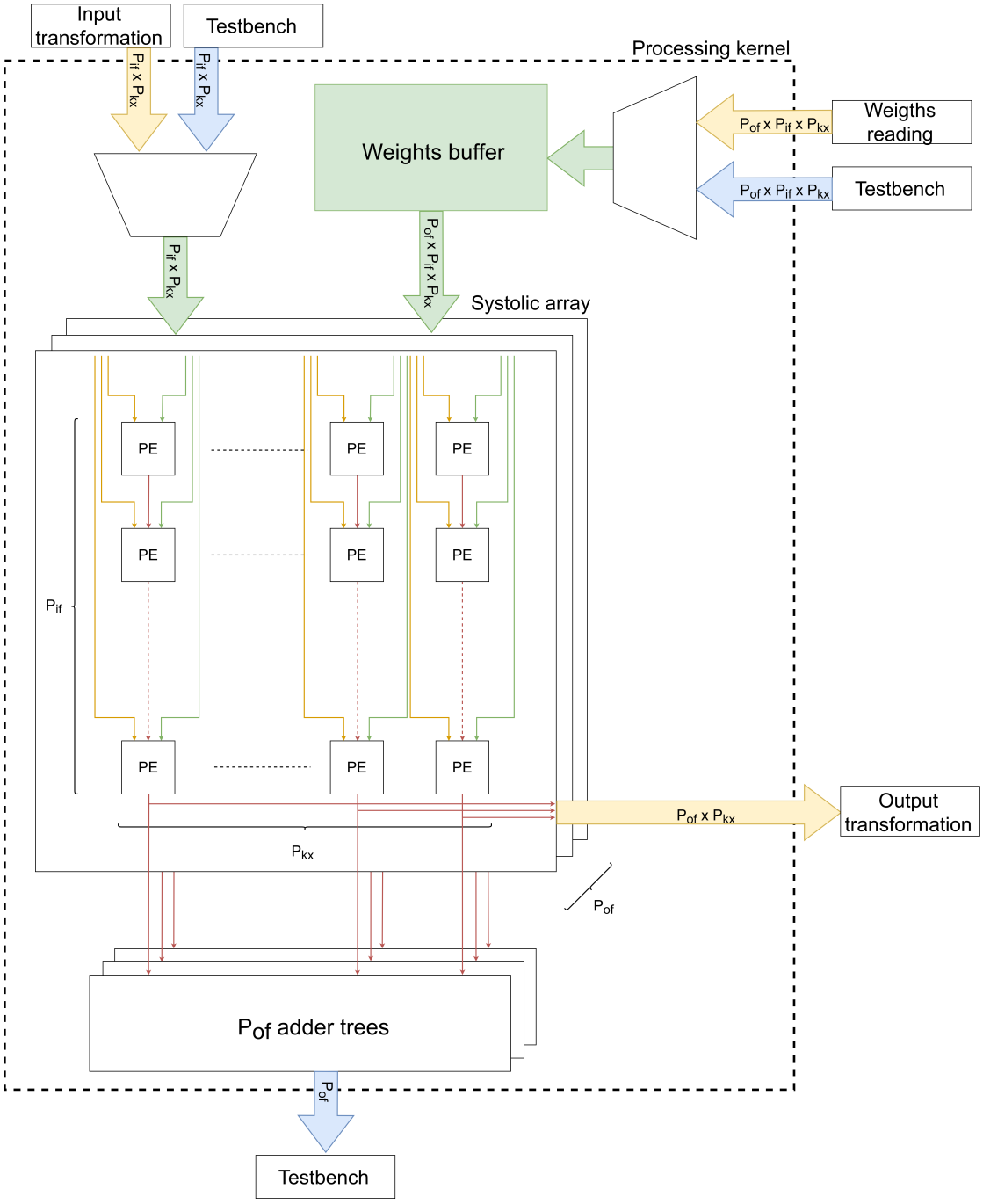


Figure 6.6: Processing kernel structure

Also the total number of multiplications performed by the code execution changes with the applied algorithm, equations 6.1 and 6.2 show how they are computed,

for the standard and the Winograd cases respectively.

$$\#\ mul\ (std) = \lceil \frac{N_{of}}{P_{of}} \rceil \times N_{ox} \times N_{oy} \times \lceil \frac{N_{kx}}{P_{kx}} \rceil \times N_{ky} \times \lceil \frac{N_{if}}{P_{if}} \rceil \times P_{of} \times P_{if} \times P_{kx} \quad (6.1)$$

$$\#\ mul\ (W) = \lceil \frac{N_{of}}{P_{of}} \rceil \times \lceil \frac{N_{ox}}{4} \rceil \times \lceil \frac{N_{oy}}{4} \rceil \times \lceil \frac{46}{P_{kx}} \rceil \times \lceil \frac{N_{if}}{P_{if}} \rceil \times P_{of} \times P_{if} \times P_{kx} \quad (6.2)$$

Approximating $\lceil \frac{N_{ox}}{4} \rceil \times \lceil \frac{N_{oy}}{4} \rceil = \frac{N_{ox} \times N_{oy}}{16}$, from the ratio of the two values it is possible to derive the multiplications saving of $F(4\times4,3\times3)$ respect to the standard convolution in the implemented design:

$$muls\ saving = \frac{\lceil \frac{N_{kx}}{P_{kx}} \rceil \times N_{ky}}{\frac{1}{16} \times \lceil \frac{46}{P_{kx}} \rceil} \quad (6.3)$$

Being in $F(4\times4,3\times3)$ the kernel dimensions equal to 3x3, the equation 6.3 becomes:

$$muls\ saving = \frac{\lceil \frac{3}{P_{kx}} \rceil \times 3}{\frac{1}{16} \times \lceil \frac{46}{P_{kx}} \rceil} \quad (6.4)$$

From this relation, it is possible to see that the ratio changes together with the unrolling parameter P_{kx}. In particular, looking at (1,2,4) as a set of possible P_{kx} values (because they are powers of 2 and values higher than 4 could be counterproductive given the size of the kernel), the possible ratio values are:

- 3.13 using $P_{kx} = 1$
- 4.17 using $P_{kx} = 2$
- 4 using $P_{kx} = 4$

Only in the case of $P_{kx} = 1$, the value is equal to the theoretical one derived in section 4.4.1. In the other two cases, the standard convolution wastes 1 multiplication each 4, because of how the 3×3 kernels are read (section 5.4). In the Winograd algorithm case, instead, inputs and weights are handled as sets of 46 elements vectors, this avoids wasting multiplications in both the cases $P_{kx} = 1$ and $P_{kx} = 2$. When $P_{kx} = 4$, 48 elements are read instead of 46 for each vector, so 2 multiplications are wasted for each 48.

Algorithm 5 Processing loop

```
if winograd then
    out_xy = (N_ox × N_oy) / 16
    MAC_loop = ⌈46 / P_kx⌉ × ⌈N_if / P_if⌉
end
else
    out_xy = N_ox × N_oy
    MAC_loop = ⌈N_kx / P_kx⌉ × N_ky × ⌈N_if / P_if⌉
end
for of = 0; of < ⌈N_of / P_of⌉; of++ do
    for xy = 0; xy < out_xy; xy++ do
        for c = 0; c < MAC_loop; c++ do
            if xy==0 then
                if winograd then
                    load P_of × P_if × P_kx weights from IT
                end
                else
                    load P_of × P_if × P_kx weights from TB
                end
            end
            if winograd then
                read P_if × P_kx inputs from IT
            end
            else
                read P_if × P_kx inputs from TB
            end
            P_of × P_if × P_kx multiplications
            accumulation of the results in P_if dimension
            if !winograd then
                accumulate results in P_kx dimension
            end
        end
        if winograd then
            send P_of × P_kx element to OT
        end
        else
            send P_of element to TB
        end
    end
end
```

Looking at the same algorithm, it is also possible to see how the unrolling levels influence the on-chip memory demand. Considering that the weights are loaded once for each iteration of the outer loop, the size of the weights buffer must be equal at least to the number of loaded weights each outer loop iteration, so it is possible to derive the following relations:

- $weights\ buffer\ size \geq weights\ width \times P_{of} \times P_{if} \times P_{kx} \times \lceil \frac{N_{kx}}{P_{kx}} \rceil \times N_{ky} \times \lceil \frac{N_{if}}{P_{if}} \rceil$ in the case of the standard convolution
- $weights\ buffer\ size \geq weights\ width \times P_{of} \times P_{if} \times P_{kx} \times \lceil \frac{46}{P_{kx}} \rceil \times \lceil \frac{N_{if}}{P_{if}} \rceil$ in the case of the Winograd convolution

In both two cases, the on-chip memory usage depends on both the unrolling parameters and the size of the parameters. However, to be feasible to synthesize the IP, the weights buffer size must not depend on the inputs, and so on their sizes. So an arbitrary constant value (*size*) must substitute the second part of both the expressions. This value limits the maximum number of eligible input channels and the kernel sizes. The final value corresponding to the on-chip memory demand, to store the weights buffer, is:

$$weights\ buffer\ size = weights\ width \times P_{of} \times P_{if} \times P_{kx} \times size$$

Where:

$$size \geq max(\lceil \frac{maxN_{kx}}{P_{kx}} \rceil \times maxN_{ky} \times \lceil \frac{maxN_{if}}{P_{if}} \rceil, \lceil \frac{46}{P_{kx}} \rceil \times \lceil \frac{maxN_{if}}{P_{if}} \rceil)$$

In the HLS code, this buffer is implemented as an array of *size* structures of $P_{of} \times P_{if} \times P_{kx}$ weights.

6.2.4 Output transformation (OT)

This kernel aims to perform the transformations of processing kernel's outputs, and then send the transformed values to the testbench.

It reads $P_{kx} \times P_{of}$ 32-bit values each iteration. These values are used to fill P_{of} array of 46 elements.

The first computational step performed by the kernel is to compute the differences of the Karatsuba algorithm to derive the results of the complex multiplications. Thus, the values can be organized in P_{of} arrays of 36 elements representing the E matrices of the Winograd algorithm.

The output transformation is computed resorting to shifts and additions, the results of this operation are P_{of} arrays of 16 elements 32-bit wide representing the 4×4 OFMaps tiles. Their values are sent to the testbench, through a stream, $P_{kx} \times P_{of}$ values each iteration.

Since the whole kernel works using 32-bit values, its logical element demand (that is also linear together with P_{of}) highly contributes to increasing the one of the whole IP.

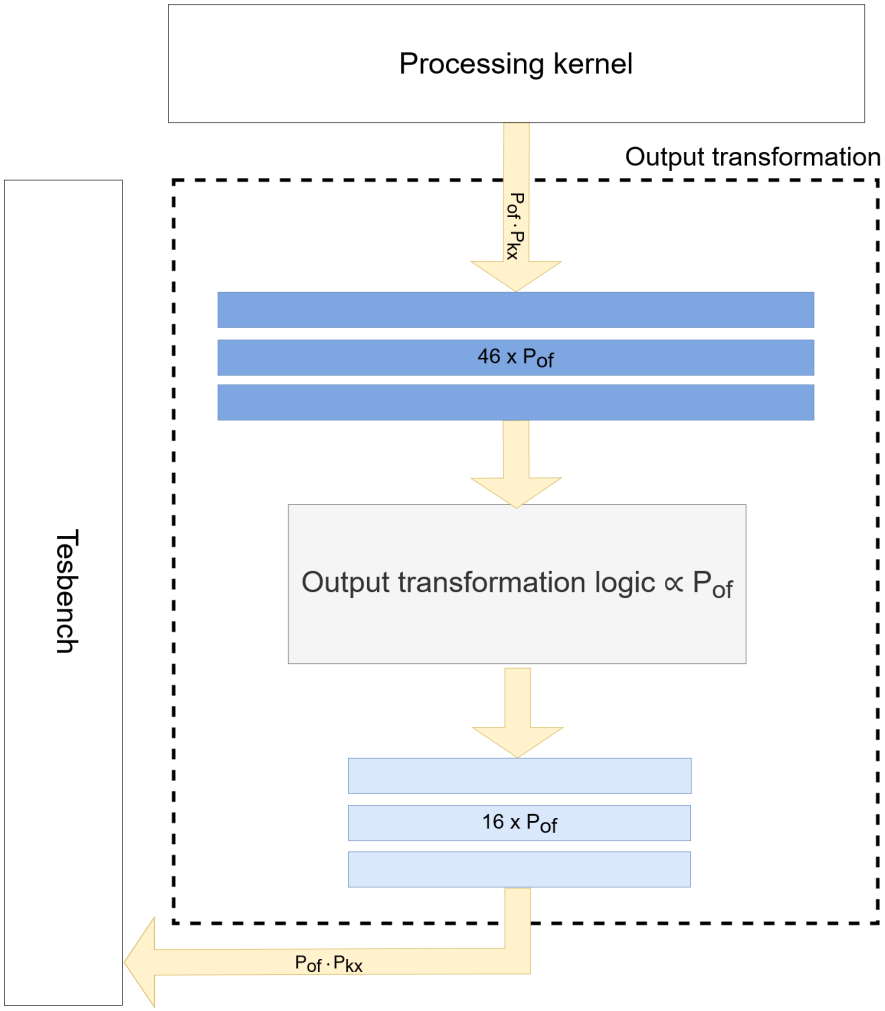


Figure 6.7: Output transformation kernel structure

Algorithm 6 shows that there are $\lceil \frac{N_{of}}{P_{of}} \rceil \times \frac{N_{ox} \times N_{oy}}{16}$ iterations of:

- $\lceil \frac{46}{P_{kx}} \rceil$ stream reading (of $P_{of} \times P_{kx}$ value)
- P_{of} arrays transformation
- $\lceil \frac{16}{P_{kx}} \rceil$ stream writing (of $P_{of} \times P_{kx}$ value)

Algorithm 6 Weights reading loop

for $of = 0;\ of < \lceil \frac{N_{of}}{P_{of}} \rceil;\ of{+}{+}$ **do**
  for $xy = 0;\ xy < \frac{N_{ox} \times N_{oy}}{16};\ xy{+}{+}$ **do**
    for $s = 0;\ s < \lceil \frac{46}{P_{kx}} \rceil;\ s{+}{+}$ **do**
      $read\ P_{of} \times P_{kx}\ element\ from\ PK$
    end
    $karatsuba\ subtractions$
    $transformation\ of\ P_{of}\ arrays$
    for $s = 0;\ s < \lceil \frac{16}{P_{kx}} \rceil;\ s{+}{+}$ **do**
      $send\ P_{of} \times P_{kx}\ element\ to\ TB$
    end
  end
end

Chapter 7

Experiments

This chapter describes firstly how the algorithms were validated through C simulations, then analyzes the reports of the synthesis of different solutions, which implement different unrolling levels, and, in conclusion, the results of the C/RTL co-simulations, which establish if the synthesized RTL solutions work correctly and estimate the solutions latencies.

7.1 C simulations

The C simulations were performed by flanking a MATLAB script with the C testbench. The MATLAB script produces random input and weights, performs the same convolution required of the *C++* code, and produces the expected results. Inputs, weights, and outputs are saved in .txt files which are then read by the testbench. The testbench vectorizes inputs and weights and sends them to the kernels. Once the kernels have produced their results and the testbench has received them, they are compared to the ones produced by the MATLAB script. Several simulations were run, to test both the standard convolution algorithm and the Complex Winograd algorithm, with different configurations of the IFMaps and weights dimensions, also using different unrolling levels. Each test gave positive results confirming that the implemented algorithm worked as expected. The two solutions were also compared between them and the error introduced by the Winograd algorithm was in line with the results observed in chapter 4 (18 as maxiumum absolute value and 1.53 as average absolute value).

7.2 C synthesis

Once the algorithm is validated, the C synthesis is used to derive an RTL implementation. Chapter 6 discusses how the unrolling levels theoretically influence the hardware demand. The graphs below show how these relations really influence the percentage of DSPs, on-chip memory and LUTs in synthesized solutions for the XILINX Zynq UltraScale+ MPSoC ZCU104. In particular, the graphs in figures 7.1, 7.2 and 7.3 show, respectively, the P_{if}, P_{of} and P_{kx} parameters effects.

In the first graph configurations, the values of P_{of} and P_{kx} are both held constant at 4, while the parameter P_{if} varies from 2 to 16. It is notable that the on-chip memory and the DSPs usage increase linearly together with the parameter under test. The LUTs usage, instead, grows slower because the output transformation logic demand depends only on the P_{of} parameter. Its usage is about 7 percent for all four configurations.

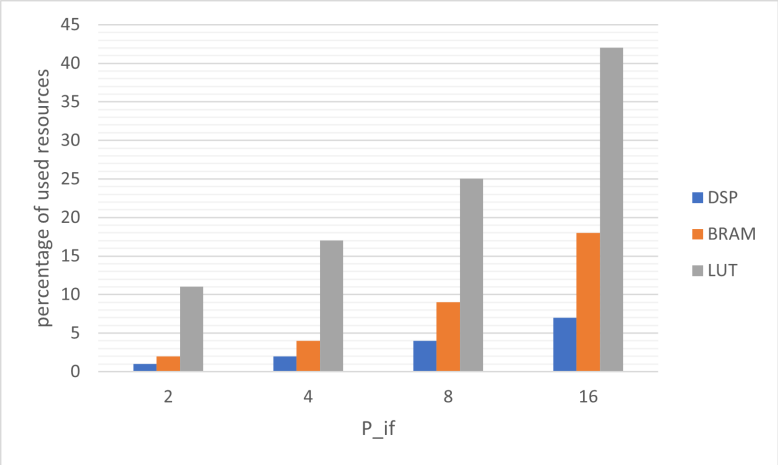


Figure 7.1: Changes in resource demand consequent to P_{if}'s change

To evaluate the changes introduced by the P_{of} variations, the P_{if} and P_{kx} parameters are held constant at 4, while the parameter P_{of} varies from 2 to 16. The synthesis results are similar to the P_if case because the two parameters influence the DSPs and memory usage in the same way and P_{of} does not influence the input transformation logic usage that is kept constant at 3% but is related to the output transformation LUTs usage.

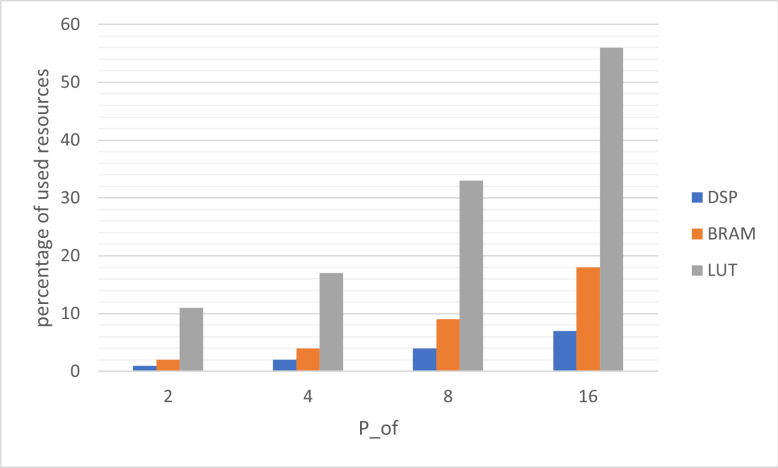


Figure 7.2: Changes in resource demand consequent to P_{of} 's change

Keeping the P_{of} and P_{if} values constant at 8 and varying the P_{kx} parameter, it is possible to notice how also in this case the use of DSP and on-chip memory grows linearly with P_{kx}, while that of the LUTs does not. In this case, however, the LUTs demand differences are less noticeable, this is because only the logic elements used in the processing kernel change together with the parameter changes, while the LUTs needed by the other three kernels are kept constant in the four configurations.

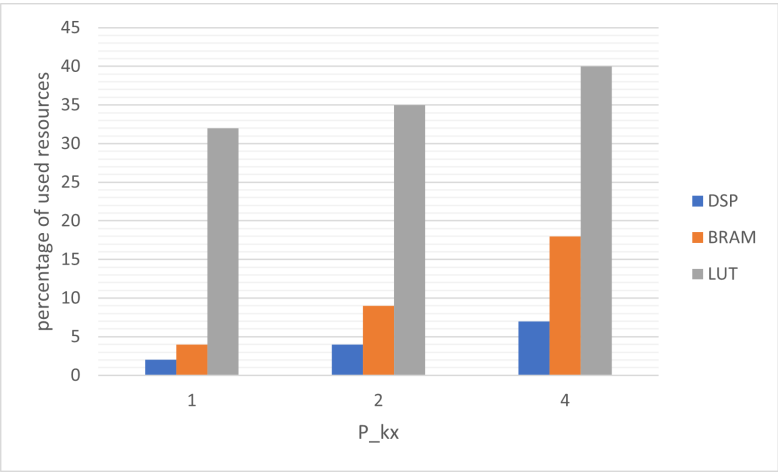


Figure 7.3: Changes in resource demand consequent to P_{kx} 's change

The results shown by the 3 graphs are in line with the theoretical relationships obtained in the previous chapter. Looking at them is also evident that the LUTs demand is the most critical value, which limits the maximum achievable level of

unrolling. Therefore, even if the number of available DSPs is 1728, their maximum usage in a synthesizable configuration is only 256 (14%), allowing a maximum of 512 multiplications in parallel (performing two multiplication in a single DSP). The table below shows the hardware usage summary of the configuration implementing $P_{if} = 16$, $P_{of} = 16$ and $P_{kx} = 2$.

DSPs	BRAM (%)	LUT (%)
256	39	89

Table 7.1: Synthesis summary of the IP using $P_{if} = 16$, $P_{of} = 16$, $P_{kx} = 2$

7.3 C/RTL co-simulations

To validate the synthesized RTL solutions, C/RTL co-simulations are run. They allow knowing if the RTL implementation is functionally identical to the *C++* code. The same testbenches used in the C simulations can be used to run the tests. Co-simulations can also be used to estimate the latencies. They return the numbers of clock cycles required to execute the run tests. Therefore, it is possible to see how the execution latency changes with the unrolling parameters and to evaluate the Winograd algorithm benefits. A C/RTL co-simulation was run for each synthesized configuration, and all the RTL implementations resulted in functionally identical to the C++ code. They were tested using a sample layer of the same size as the Conv2_1 layer of the ResNet18 [21], so: $N_{ix} = N_{iy} = 56$; $N_{if} = N_{of} = 64$; $N_{kx} = N_{ky} = 3$; $stride = 1$; $padding = 1$. The obtained latencies were then compared to derive the following considerations.

How it is possible to imagine, the higher unrolling factors values are, the lower the overall latency is, because of the operations parallelization. This relation is shown in the graph in figure 7.4, where both the standard and Winograd convolutions are analyzed.

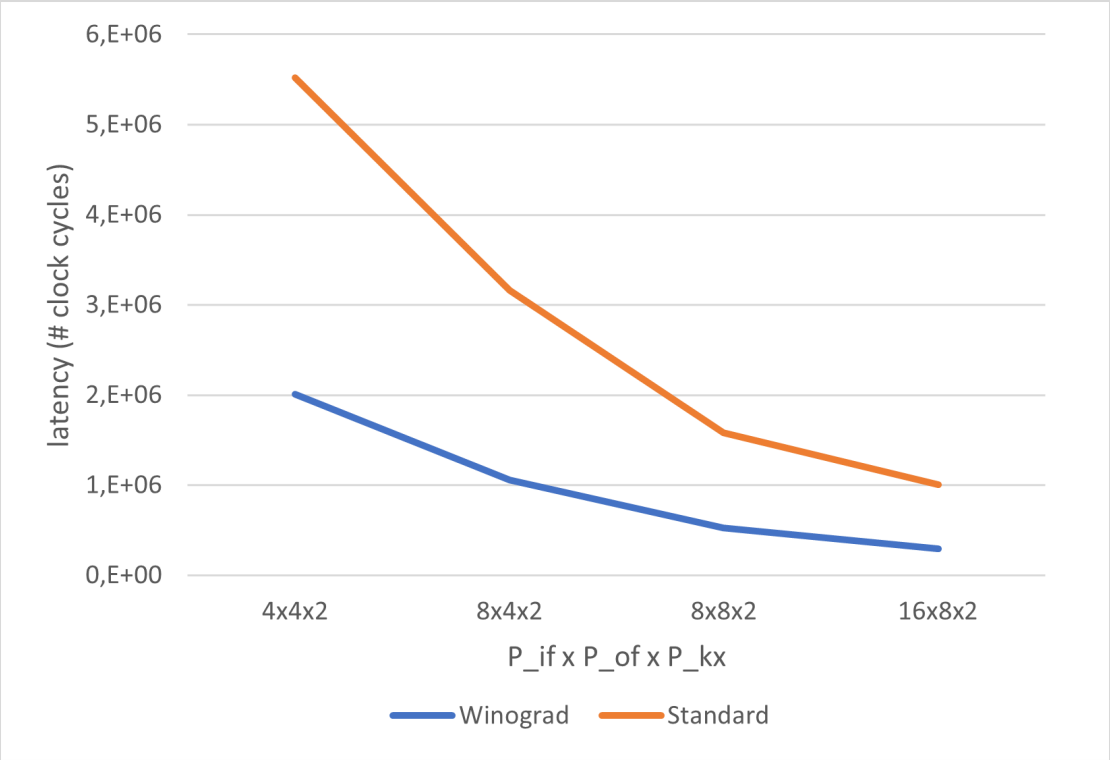


Figure 7.4: Relation between unrolling parameters and convolutions latencies

Table 7.2, instead, shows how the ratio between the latencies of the two different algorithms is higher as the unrolling parameters increase. Therefore, it becomes closer to the multiplication ratio, which, using $P_{kx} = 2$, is equal to 4.17 as derived in section 6.2.3. The two ratios are more different in smaller configurations because there are fewer multiplications performed simultaneously, so the contribution to the latency of the rest of the circuit is more noticeable.

$P_{if} \times P_{of} \times P_{kx}$	latencies ratio
$4 \times 4 \times 2$	2.74
$8 \times 4 \times 2$	2.99
$8 \times 8 \times 2$	3.00
$16 \times 8 \times 2$	3.44

Table 7.2: Relation between the unrolling parameters and the latencies ratio

Lastly, table 7.3 shows the latency of three configurations, which have different P_{kx} values but they make use of the same DSPs number. It is interesting to see how the ratio between the two latencies changes together with the P_{kx} parameter.

This is because the multiplications are the major contribute to the overall system latency.

$P_{if} \times P_{of} \times P_{kx}$	multiplications ratio	latencies ratio
$8 \times 4 \times 1$	3.13	2.11
$4 \times 4 \times 2$	4.17	2.74
$4 \times 2 \times 4$	4	2.66

Table 7.3: Relation between P_{kx} and the latencies ratio

Chapter 8

Conclusions

8.1 Summing-up and further works

This thesis work proposes a design of an IP able to perform the convolution operation for a quantized neural network. Taking the convolutional kernel of [17] as the baseline, several optimizations were applied to maximize resource utilization and reduce the computational cost.

An 8-bit fixed-point quantization is applied to reduce the memory footprint. This allows also exploiting the parallel computation mapping two multiplications in a single DSP slice.

To further exploit the parallelization, a three-dimensional unrolling is implemented, increasing the throughput of the system and resource utilization. The three unrolling levels are scalable allowing control of the latency and resource demand, giving the system certain flexibility, typical of FPGA solutions.

Complex and RNS representations of the Winograd algorithms $F(2 \times 2, 3 \times 3)$, $F(4 \times 4, 3 \times 3)$ and $F(6 \times 6, 3 \times 3)$ were analyzed, to reduce the computational cost of the convolution, with a limited numerical error. Complex Winograd $F(4 \times 4, 3 \times 3)$ was preferred among the possible solutions, to be implemented in the design. It introduced a multiplications reduction of 3.13× compared to the standard convolution.

The solution was synthesized for a XILINX Zynq UltraScale+ MPSoC ZCU104 FPGA and then tested. The collected results confirmed the benefits of the implemented optimizations. However, the results also demonstrated that the Winograd awareness greatly affects the logic resources demand, limiting the maximum level reachable by the parallelization.

As a next step, the developed IP has to be implemented and validated in hardware. After that, an evaluation of the ResNet-18 inference, for both algorithms, will be able to quantify the real advantage introduced by the Winograd algorithm

in terms of latency.

The RNS solution was not the best option in this project, however, it would be interesting to study the possibility of using it in Winograd algorithms applied on larger input tiles and/or with larger kernels because they should ensure a greater reduction in complexity, and the CRT operation drawbacks can be overcome.

Bibliography

[1] Jiuxiang Gu et al. «Recent advances in convolutional neural networks». In: *Pattern Recognition* 77 (2018), pp. 354–377 (cit. on p. 7).

[2] Qing Li, Weidong Cai, Xiaogang Wang, Yun Zhou, David Dagan Feng, and Mei Chen. «Medical image classification with convolutional neural network». In: *2014 13th international conference on control automation robotics & vision (ICARCV)*. IEEE. 2014, pp. 844–848 (cit. on p. 7).

[3] Inad Aljarrah and Duaa Mohammad. «Video content analysis using convolutional neural networks». In: *2018 9th International Conference on Information and Communication Systems (ICICS)*. IEEE. 2018, pp. 122–126 (cit. on p. 7).

[4] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. «Convolutional neural networks for speech recognition». In: *IEEE/ACM Transactions on audio, speech, and language processing* 22.10 (2014), pp. 1533–1545 (cit. on p. 7).

[5] Jiantao Qiu et al. «Going deeper with embedded fpga platform for convolutional neural network». In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 26–35 (cit. on p. 7).

[6] S Winograd. «Arithmetic complexity of computations, CBMS-NSF Reg». In: *Conf. Series in Applied Math., Philadelphia*. 1980 (cit. on pp. 7, 20).

[7] A. Lavin and S. Gray. «Fast Algorithms for Convolutional Neural Networks». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2016, pp. 4013–4021. DOI: 10.1109/CVPR.2016.435. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.435 (cit. on pp. 7, 24, 37).

[8] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. «Quantized neural networks: Training neural networks with low precision weights and activations». In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898 (cit. on p. 7).

[9] Lingchuan Meng and John Brothers. «Efficient Winograd Convolution via Integer Arithmetic». In: *ArXiv* abs/1901.01965 (2019) (cit. on pp. 7, 24).

[10] Zhi-Gang Liu and Matthew Mattina. «Efficient Residue Number System Based Winograd Convolution». In: *ECCV*. 2020 (cit. on pp. 7, 21, 24).

[11] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. «Efficient processing of deep neural networks: A tutorial and survey». In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329 (cit. on p. 9).

[12] Li Deng. «The mnist database of handwritten digit images for machine learning research». In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142 (cit. on p. 17).

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «Imagenet: A large-scale hierarchical image database». In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cit. on p. 17).

[14] *XILINX official site*. URL: `https://www.xilinx.com/products/boards-and-kits/zcu104.html` (visited on 03/23/2022) (cit. on p. 19).

[15] Barbara Barabasz, Andrew Anderson, Kirk M Soodhalter, and David Gregg. «Error analysis and improving the accuracy of Winograd convolution for deep neural networks». In: *ACM Transactions on Mathematical Software (TOMS)* 46.4 (2020), pp. 1–33 (cit. on p. 20).

[16] Javier Fernandez-Marques, Paul Whatmough, Andrew Mundy, and Matthew Mattina. «Searching for winograd-aware quantized networks». In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 14–29 (cit. on pp. 20, 29).

[17] Dong Wang, Jianjing An, and Ke Xu. «PipeCNN: An OpenCL-based FPGA accelerator for large-scale convolution neuron networks». In: *arXiv preprint arXiv:1611.02450* (2016) (cit. on pp. 23, 37, 63).

[18] Dong Nguyen, Daewoo Kim, and Jongeun Lee. «Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs». In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 890–893 (cit. on p. 38).

[19] HT Kung and Charles E Leiserson. «Systolic arrays (for VLSI)». In: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for industrial and applied mathematics. 1979, pp. 256–282 (cit. on p. 49).

[20] Xinheng Liu, Yao Chen, Cong Hao, Ashutosh Dhar, and Deming Chen. «WinoCNN: Kernel sharing Winograd systolic array for efficient convolutional neural network acceleration on FPGAs». In: *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2021, pp. 258–265 (cit. on p. 49).

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep residual learning for image recognition». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 60).