POLITECNICO DI TORINO

Hardware Acceleration for Neural Networks

by

Sahar Eslami

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the
Department of Electronics and Telecommunications Engineering

October 2018

Declaration of Authorship

I, Sahar Eslami, declare that this thesis titled, 'Hardware Acceleration for Neural Networks' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

The purpose of this project is to optimize a convolutional neural network algorithm called YOLO via high-level synthesis based on an FPGA board.

The work is divided into two main parts, the first part is the transformation of YOLO C++ tensor-based code into an implementation for Vivado HLS and Catapult compiler; The second part is optimizing the network in the two HLS tools for hardware implementation and compare the power and utilization for the two cases.

The HLS tools which are Vivado_hls and Catapult, use a series of steps to generate the hardware. Vivado is the third tool used in this project to compare the results between these different cases.

In this work, C++ language is used to write the code, Vivado _ hls and Catapult are used to transfer the code into hardware, VHDL is used to package the IP and the hardware.

Acknowledgements

I would like to thank my supervisor, Prof. Luciano Lavagno for his support and motivation throughout the course of this work. He provided me the opportunity to improve my technical skills and the resources to fulfill this task.

I also thank Dr.Ma Liang, for all his suggestions and useful instructions. Also other classmates in DET department, including Dr.Shan Junnan and Dr. Arslan Arif.

I am thankful to my family who have always been a source of motivation for me in different steps of my life.

Dedicated to my family.

Contents

List of Figures

Abbreviations

CNN	**C**onvolutional **N**eural **N**etwork
FC	**F**ully **C**onnected
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
II	**I**nitiation **I**nterval
IOU	**I**ntersection **O**ver **U**nion
NMS	**N**on **M**axima **S**uppression
RAM	**R**andom **A**ccess **M**emory
RTL	**R**egister **T**ransfer **L**evel
YOLO	**Y**ou **O**nly **L**ook **O**nce

Chapter 1

Intoduction

1.1 Deep Learning

Deep learning is a branch of machine learning based on a set of algorithms that learn to represent the data.

One of the promises of deep learning is that they will substitute hand-crafted feature extraction. The idea is that they will "learn" the best features needed to represent the given data.

Deep learning models are formed by multiple layers. A network with more than 2 hidden layers is already a Deep Model.

1.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a deep learning algorithm that is composed of layers which filter(convolve) the inputs to get useful information. A CNN is very similar to an ordinary neural network, meaning that it consists of neurons with learning weights and biases. The filters of a CNN are adjusted automatically to extract the most useful information from the raw image pixels on one end and turn that information into a class score on the other end.

Therefor just like a neural network, a CNN receives an input and transforms it through a set of hidden layers. Each hidden layer consists of a set of neurons. The neurons in a

single layer do computations completely independently from other layers and they are not connected to other neurons in the same layer.

The last layer or the output layer is where the class scores are represented.

So why use CNNs instead of regular neural networks? The answer is that in a neural network each single neuron is connected to all the neurons in the previous layer. This will make the neural network too complicated for image inputs.

As an example, with an input image of size 16x16x3 (height=16, width=16, depth=3) each single neuron in a first hidden layer of a regular neural network must have 16*16*3 equal to 768 weights. Although this seems like a good number but this fully connected structure can not scale to larger images.

So CNNs have the benefit of assuming that the inputs are strictly images and with that assumption they can apply some sensible changes to the regular neural networks. For example a neural network receives input as a vector, while a CNN has neurons arranged in 3 dimensions: height, width, depth.

the neurons in a CNN will only be connected to a small region of the previous layer, but to the full depth and this will save huge amount of computations. Also the full image will be converted to a single vector of class scores for the last layer.

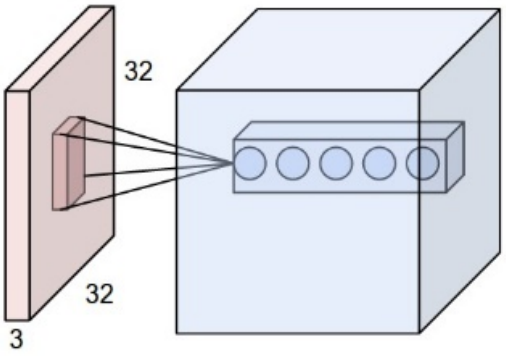

FIGURE 1.1: Neurons in each layer

As you can see in the figure 1.1, there are 5 neurons along the depth, all looking at the same region in the input.

You can see a comparison between a regular neural network and a CNN in the following picture:

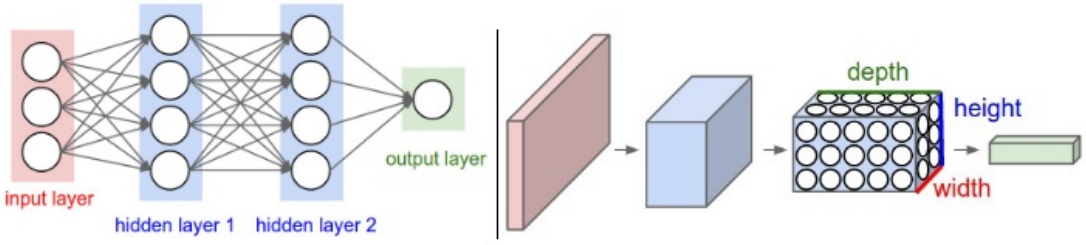

FIGURE 1.2: Neural Network and CNN comparison

The most common layers of a CNN are the following:

Convolution Layer
Max/Average Pooling Layer
Fully Connected Layer
Relu, Tanh, Sigmoid Layer (Non-Linearity Layers)
Softmax, Cross Entropy, SVM, Euclidean (Loss Layers)

So in summary a CNN in the simplest case is a series of layers that "convole" the input volume into the output volume.
Each layer receives a 3D input volume that is transformed into a 3D output volume through different functions.
Each layer may or may not have parameters. For example convolutional and FC layers do, but relu and pooling layers don't.

In the following some of these layers will be explained in more details.

1.2.1 Convolutional Layer

As mentioned before, a convolutional layer extracts the different features of the input. They are the core building block of the CNN with the heaviest computational parts and are used to find a specific patch on an image.

The important parameters of this layer are:
F: Number of filters on the convolution layer
kW/kH: Kernel Width/Height (Normally we use square images, so kW=kH)
H/W: Image height/width (Normally H=W)
H'/W': Convolved image height/width (Remains the same as input if proper padding is used)

Stride: Number of pixels that the convolution sliding window will travel.
Padding: Zeros added to the border of the image to keep the input and output size the same.
Depth: Volume input depth (for example if the input is a RGB image depth will be 3)

The output size will be computed as follows:

Output height: (H-F+2Padding)/Stride + 1
Output width: (W-F+2Padding)/Stride + 1
Output depth: F

A common setting of the parameters is : F=3, Stride=1 and Padding=1.

In figure 1.3 you can see an example of the convolution process.

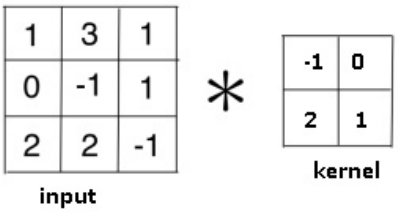

FIGURE 1.3: Convolution

The input elements must be multiplied element by element with the filter(kernel) and the results must be summed.

This Process will result in extracting some specific features from the input image. (Figure 1.4)

By default when we're doing convolution we move our window one pixel at a time (stride=1), but some times in convolutional neural networks we want to move by more than one pixel. For example on pooling layers with kernels of size 2 we will use a stride of 2.

Figure 1.6 illustrates two examples on input and output volume calculations. The input size is 5 with padding equal to 1. On the left neurons stride across the input with stride of 1 and therefor with F=3 the output size will be: (5 - 3 + 2)/1+1 = 5
On the right the neurons stride across the input with stride of 2 and the output size will be: (5 - 3 + 2)/2+1 = 3.

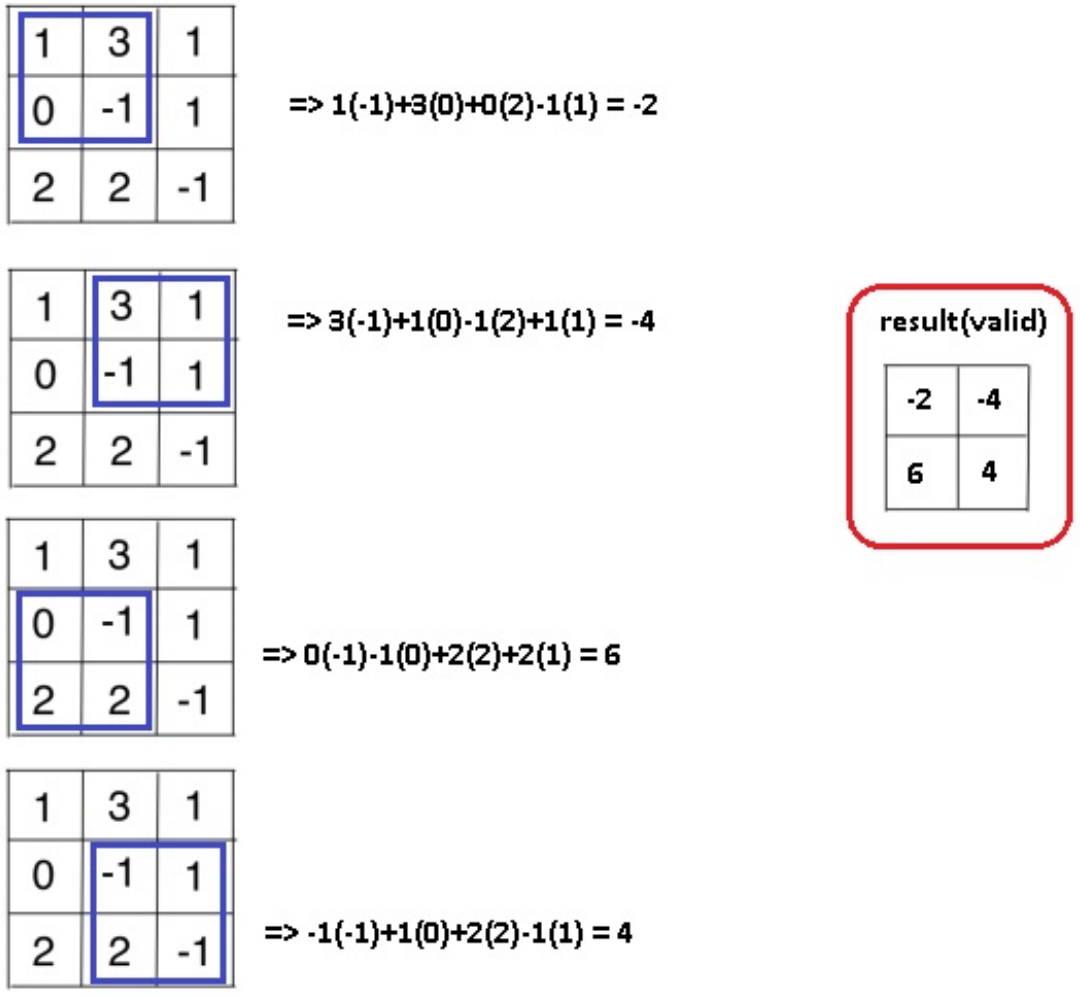

FIGURE 1.4: Convolution Process

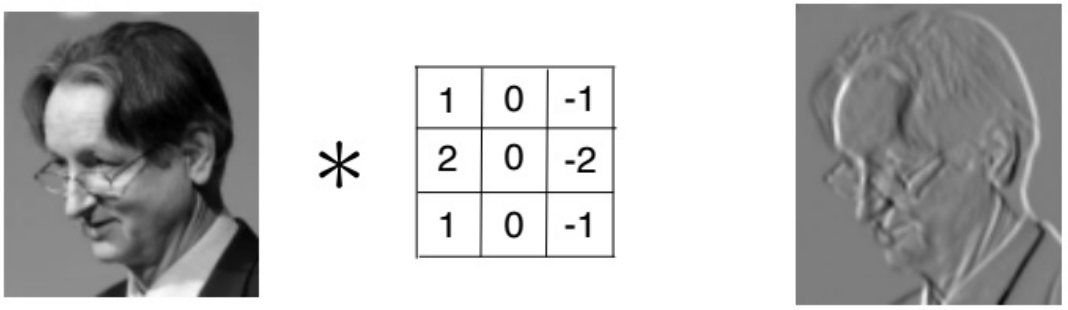

FIGURE 1.5: Feature Extraction

1.2.2 Relu Layer

The Relu layer will change all negative elements to zero while retaining the value of the positive elements. No spatial/depth information is changed.

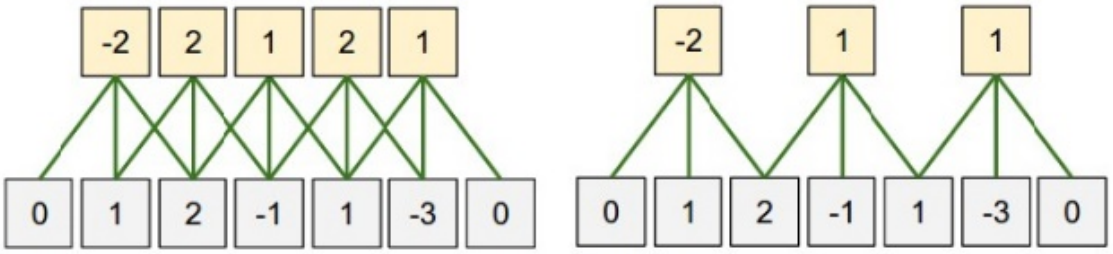

FIGURE 1.6: Neurons spatial arrangments

Considering the neural networks, it's just a new type of Activation function, but Easy to compute the forward/backward propagation.

```
>> A = randn(4,4)

A =

   -0.1961    1.5877   -0.2437    0.1049
    1.4193   -0.8045    0.2157    0.7223
    0.2916    0.6966   -1.1658    2.5855
    0.1978    0.8351   -1.1480   -0.6669

>> relu_A = max(0,A)

relu_A =

         0    1.5877         0    0.1049
    1.4193         0    0.2157    0.7223
    0.2916    0.6966         0    2.5855
    0.1978    0.8351         0         0

fx >>
```

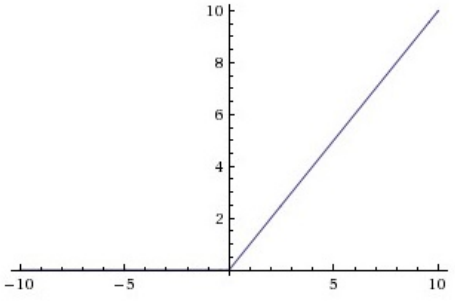

FIGURE 1.7: Relu function

1.2.3 Pooling Layer

The pooling layer, is used to reduce the spatial dimensions, but not depth, usually placed between successive convolutional layers. Basically this is what we can gain:
1. By having less spatial information we gain computation performance.
2.Less spatial information also means less parameters, so less chance to over-fit.

If we assume that the input for the pooling layer has the following parameters:

Input height: H
Input width: W
Input depth: D

With the spatial extent F, and stride S, then the output size will be computed as follows:

Output height: (W-F)/S+1

Output width: (H-F)/S+1

Output depth: D

On the diagram bellow you can see the most common type of pooling: the max-pooling layer, which slides a window, like a normal convolution, and gets the biggest value on the window as the output.

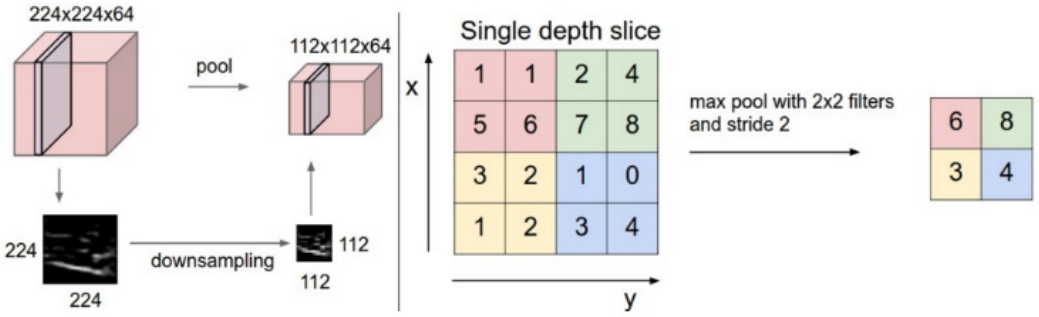

FIGURE 1.8: Max-pooling Layer

In addition to max-pooling, the pooling layer can perform other functions as well, such as average pooling or l2-norm pooling.

Two common parameter sets for the pooling layer are: F=3, S=2 (also called overlapping pooling) and more commonly F=2, S=2.

1.2.4 Fully Connected Layer

In the fully connected layer every neuron has full connections to all neurons in the previous layer, as opposed to the convolutional layer in which the neurons are connected only to a local region in the input.

Therefor a fully connected layer works exactly like a regular neural network.

1.3 CNN Architectures

Common architectures of CNN consist of a several convolutional-relu layers that are followed by pooling layers. This pattern well continue until the image is small enough to move to an FC layer. The very last FC layer maintains the output such as class scores.

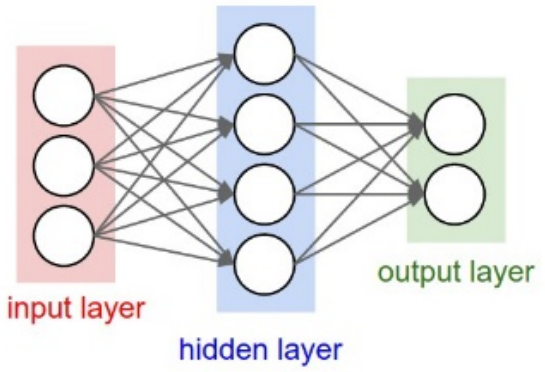

FIGURE 1.9: Fully connected layer

The CNN we are working on in this project is called YOLO. In the next chapter YOLO architecture will be fully discussed.

Chapter 2

Computer Vision and YOLO

In this chapter, the concepts of YOLO and object detection algorithms will be detailed respectively. In this project, the algorithm of YOLO is used for synthesis in Vivaso HLS and Catapult and therefor, a description of the algorithm's performance is necessary.

2.1 Introduction

Understanding the contents of an image or an image region is one of the core problems of the computer vision, fundamental to image and scene understanding. There are three different concepts related to computer vision :

Image classification
Classification with localization
Object detection

2.1.1 Image Classification

In image classification problems, the goal is to assign the input image one or more labels from some predefined set of categories, for example the set of all animals. Therefor an image classifier outputs only the class of the subject detected. The machine learning approach to image classification is a two-stage pipeline. The first step of the pipeline corresponds to extraction and encoding of meaningful features from the image pixels.

The second step performs image classification in the space of features extracted from the image.

FIGURE 2.1: Image classification

2.1.2 Classification with Localization

A classification with localization is one step forward where the algorithm defines also a "bounding box" to localize the detected object. Localizing objects is aimed with regression. Regression is about returning a number instead of a class. Normally another fully connected layer is attached to the last convolutional layer.

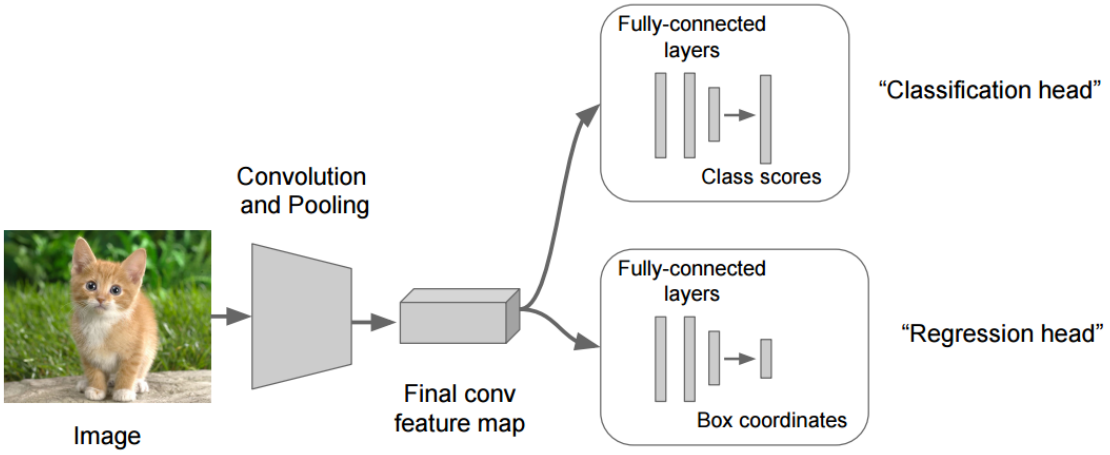

FIGURE 2.2: Image localization

Now in training set, each example contains not only the class label but also four additional bounding box numbers:

bx - x coordinate of mid point of BB

by - y coordinate of mid point of BB

bw - width of BB

bh - height of BB

Now, our convolution neural network will learn to predict both class of object as well as its bounding box.

2.1.3 Object Detection

An object detection refers to the fact that there are now multiple subjects in the picture to be detected. The algorithm has to detect all subjects with their bounding boxes.

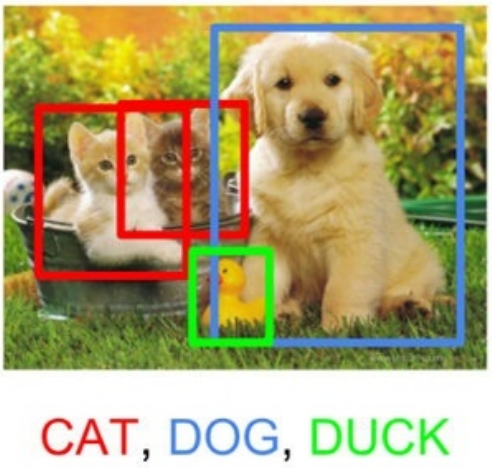

FIGURE 2.3: Object detection

2.2 YOLO

YOLO (You Only Look Once) is a state-of-the-art, real-time object detection system, detecting objects on the Pascal VOC 2012 dataset. It can detect the 20 Pascal object classes: person

bird, cat, cow, dog, horse, sheep

aeroplane, bicycle, boat, bus, car, motorbike, train

bottle, chair, dining table, potted plant, sofa, tv/monitor

YOLO applies a single neural network to the full image, which is a totally different approach with respect to prior networks. Therefor YOLO is extremely fast, more than 1000x faster than R-CNN and 100x faster than Fast R-CNN. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities.

2.2.1 Main Idea

The idea of this detector is that you run the image on a CNN model and get the detection on a single pass. First the image is re-sized to 448x448, then fed to the network and finally the output is filtered by a Non-max suppression algorithm.

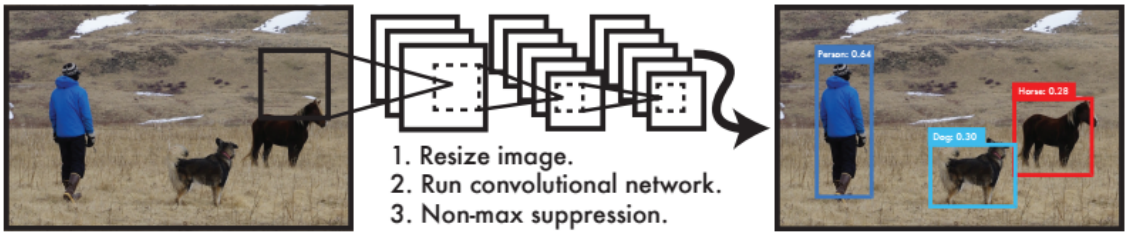

FIGURE 2.4: YOLO main idea

2.2.2 YOLO Architecture

YOLO has a different architecture for training and detection.

The training network consists of the first 20 convolutional layers of the architecture in figure 2.5, followed by an average-pooling layer and a fully connected layer. The network is trained for approximately a week and a single crop top-5 accuracy of 88 percent on the ImageNet 2012 validation set is achieved.

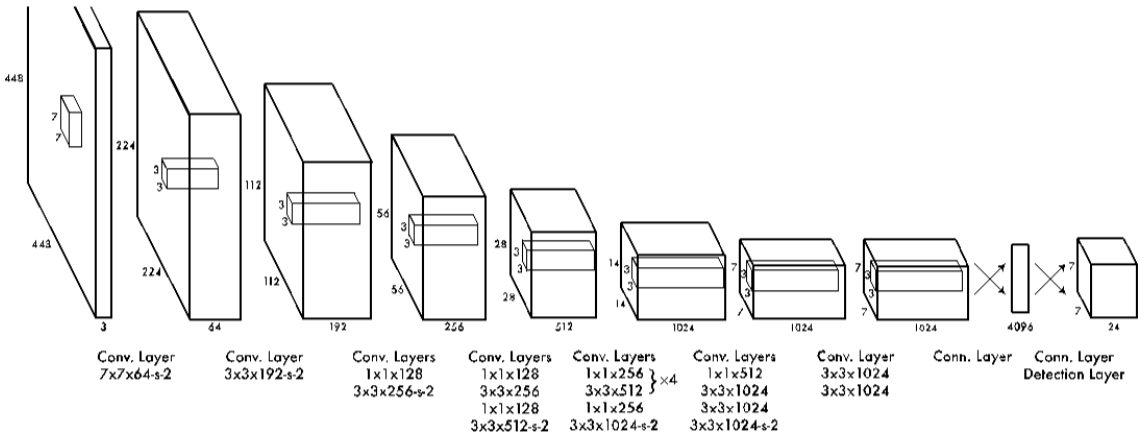

FIGURE 2.5: YOLO architecture

To convert the network for detection, four convolutional layers and two fully connected layers with randomly initialized weights are added. Since detection often requires fine-grained visual information, the input resolution of the network is increased from 224 224 to 448 448.

2.2.3 YOLO Tiny Model

For the purpose of this project, YOLO tiny model is used. The tiny version is composed of 9 convolution layers with leaky relu activations. The output of this model is a tensor batch size 7x7x30. In this tensor the following information is encoded:

2 Box definitions: (consisting of: x,y,width,height,"is object" confidence)
20 class probabilities (only considered if the "is object" confidence is high)

Tensor = S.S.(B.5 + C)

Where: S: Tensor spatial dimension (7 on this case)
B: Number of bounding boxes (x,y,w,h,confidence)
C: Number of classes

What this 7x7 tensor represents?
This 7x7 tensor can be considered as a 7x7 grid representing the input image, where each cell of this tensor will hold the 2 box definitions and 20 class probabilities.

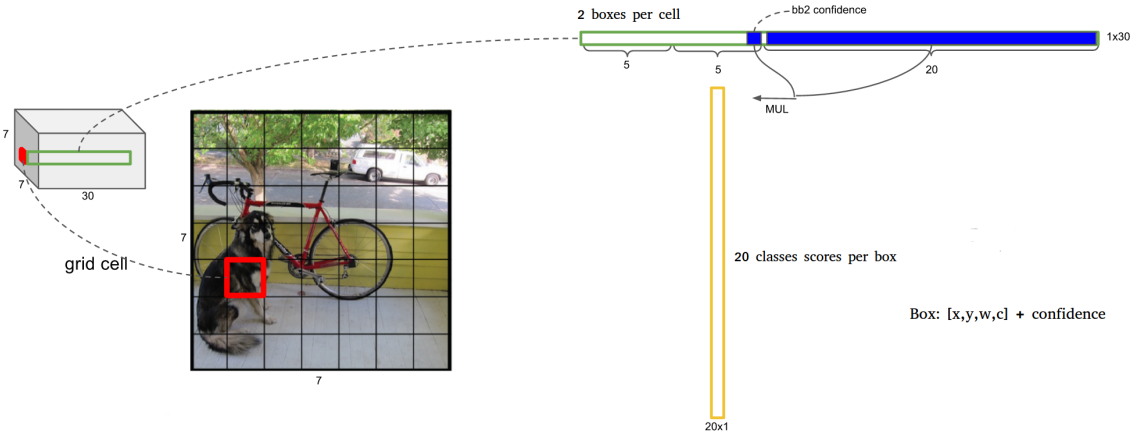

FIGURE 2.6: Tensor information

Here it's also useful to say that each cell has the probability to be one of the 20 classes. (And each cell has 2 bounding box)

this information with the fact that each bounding box has the information if it's below an object or not will help to detect the class of the object.

The logic is that if there was an object on that cell, we define which object by using the biggest class probability value from that cell.

2.2.4 Intersect over Union (IoU)

It's a method used to evaluate how well an object detection output is related to some ground truth, the IoU is normally used during training and testing by comparing how the bounding box given during prediction overlaps with the ground truth (training/test data) bounding box.

Calculating the IoU is simple we basically divide the overlap area between the boxes by the union of those areas.

2.2.5 Non-Maxima Suppression (NMS)

During prediction time (after training) you may have lots of box predictions around a single object. The NMS algorithm will filter out those boxes that overlap each other by some threshold.

FIGURE 2.7: IOU examples

In order to achieve this:

1. YOLO first detects the bounding box with the highest probability among all the bounding boxes.

2. All the remaining bounding boxes which have a high IOU with the chosen box in first phase will be omitted.

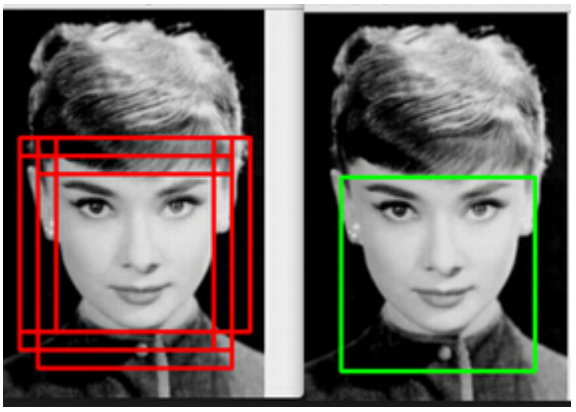

FIGURE 2.8: NMS Algorithm

Chapter 3

Vivado_HLS v.s Catapult

3.1 Background

In this thesis project, the YOLO algorithm source code which is written as tensor-based C++ code is optimized for hardware implementation. Since using Hardware Description Language(HDL) is a very complex procedure, and its consumes a lot of time for the designers to perform register transfer level(RTL), High Level Synthesis(HLS) is used to facilitate the hardware development process.

HLS is the process of transforming an algorithmic description(C/C++) of a desired behavior into hardware implementation(VHDL). In order to do that, the input code is analyzed and is then scheduled to generate RTL. This method has the advantage for the designer to use a higher level implementation of the algorithm and avid the RTL process.

The HLS process consists of the following stages:
Resource allocation
Scheduling
Binding
RTL generation

In this thesis project, Vivado HLS and Catapult are the two tools used to perform HLS for the YOLO algorithm. In this chapter each of these tools and the optimization methods are described.

3.2 Vivado_HLS

3.2.1 Overview of Vivado_HLS

The Xilinx Vivado_HLS is an HLS tool that accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx programmable devices without the need to manually create RTL.

There are several steps to be performed in Vivado HLS:
1. Create a new project
2. Designing the IP core which is developed in C++ code.
3. Test bench design to perform the C simulation and test the functionality of the algorithm
4. Prepare the IP core for synthesis: for this step the top-level function must be specified which is the sub-function below main.
5. Verify the RTL implementation: this step also uses the C test bench to verify the output of the top-level function and returns zero to main() if the RTL is functionally identical.

3.2.2 Vivado_HLS Optimization Methods

There are several methods to optimize a design using Vivado HLS including:
Inserting directives
Optimizing latency
Optimizing throughput

These different techniques result in a better hardware implementation of our design.

3.2.2.1 Inserting Directives

Inserting directives is a method to perform different architectural optimization on the IP core. To apply any directive we must first choose the element in the information panel and select "add directives".

There are mainly two categories to be optimized: Arrays and loops. ARRAY_MAP, ARRAY_PARTITION, ARRAY_RESHAPE, INTERFACE are the directives mostly used to optimize arrays.

For each loop inside the top-level function, by using add directives, we can apply ALLOCATION, DATAFLOW, DEPENDENCE, LOOP_FLATTEN, LOOP_MERGE, LOOP_TRIPCOUNT, PIPELINE, UNROLL,etc..after applying directives respectively, the performance may improve.

3.2.2.2 Loop_unroll

Syntax:
#pragma HLS unroll factor= <N> region
If factor is not specified, the loop is fully unrolled.
region: An optional keyword that unrolls all loops within the body (region) of the specified loop.

By default loops are kept rolled in C++ functions. In this case synthesis creates the logic for one iteration of the loop and the RTL performs this logic for each iteration of the loop and this is repeated for the number of iteration defined by the loop induction variable. Now with the UNROLL directive the loops can be partially or fully unrolled. If the loop is unrolled fully, a separate copy of the loop is created in RTL for each iteration and therefor the whole loop can run concurrently.
Here is an example:

```
for(i=0; i<5; i++)
{
a[i] = b[i] * [i];
}
```

Without the directive, this loop requires one multiplier and each RAM block can be single-port because each iteration is executed in one clock cycle. This implementation needs six clock cycles.

Now let's say we unroll this loop by a factor of 2, then we have two reads and two writes in the same clock cycle, which require two multipliers and dual port RAM hardware, but the advantage is that the implementation need takes 3 clock cycles to complete.

3.2.2.3 Loop_flatten

Syntax:
#pragma HLS loop_flatten off

off: If this option is active the current loop will not be flattened, while the other loops in the specified location get flattened.

In Vivado HLS, the innermost loop is flattened automatically. Since in RTL implementation it takes one clock cycle to move from an outer loop to an inner loop and another clock cycle for the opposite, loop flattening can save clock cycles by allowing nested loops into a single loop hierarchy.

As an example, if we have the following nested loops:
Outerloop: while(j<50){
Innerloop: while(i<3){ //1 cycle to enter inner loop
...
LOOP_BODY
...
} // 1 cycle to exit inner
}

In this example, if loop flattening is not used 100 additional clock cycles are needed to move between the inner and outer loops.

While this is a very efficient method to reduce latency, we must take into account that only perfect and semi-perfect loops can be flattened:
Perfect loop nests:
1.Only the innermost loop has loop body content.
2.There is no logic specified between the loop statements.
3.All loop bounds are constant.
Semi-perfect loop nests:
1.Only the innermost loop has loop body content.
2.There is no logic specified between the loop statements.
3.The outermost loop bound can be a variable.

3.2.2.4 Loop_merge

Syntax:

#pragma HLS loop_merge force

force: this keyword forces the merge even if Vivado HLS produces a warning.

This directive merges loops into a single to reduce the overall latency. As en example:

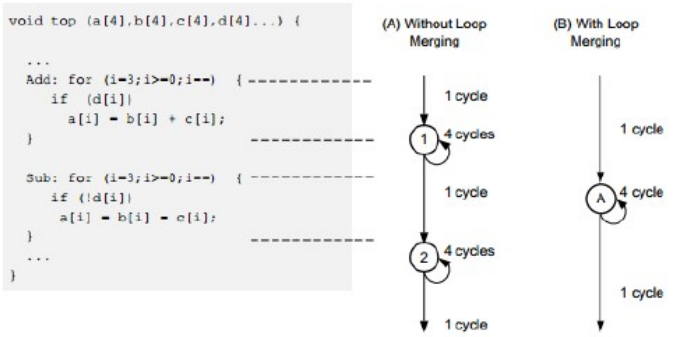

FIGURE 3.1: Loop merging

Here we can see the loop needs 11 clock cycles by default, but after the loop merging, the clock cycles are reduced to 6.

We must take into account that to apply this directive, the loop bounds MUST have the same value if they are variables, and if the loop bounds are constant, the maximum bound value is used for the merged loop.

3.2.2.5 Dataflow

Syntax:

#pragma HLS dataflow

In C++ code, all the sub functions and loops are executed in a series of sequential blocks. These blocks might have data dependencies. Therefor each block such as a function or a loop must wait for example for all the read/write operations, if it needs to access arrays.

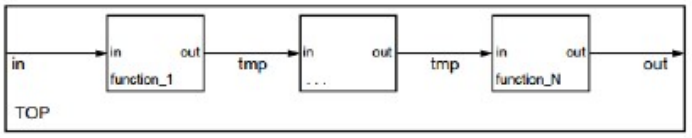

FIGURE 3.2: Sequential tasks inside top function

Now Vivado HLS aim is to reduce latency and improve concurrency. The dataflow directive examines these data dependencies between the blocks and tries to execute these blocks in parallel if the blocks can start before the previous task has finished executing. Therefor the overall latency is reduced and the throughput is improved.
The dataflow directive applies the parallelism by creating a channel based on pingpong RAMs or FIFOs.

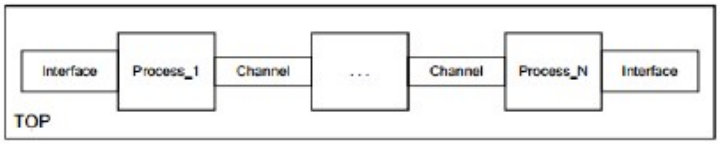

FIGURE 3.3: Parallel tasks inside top function

Let's consider the following picture as an example:

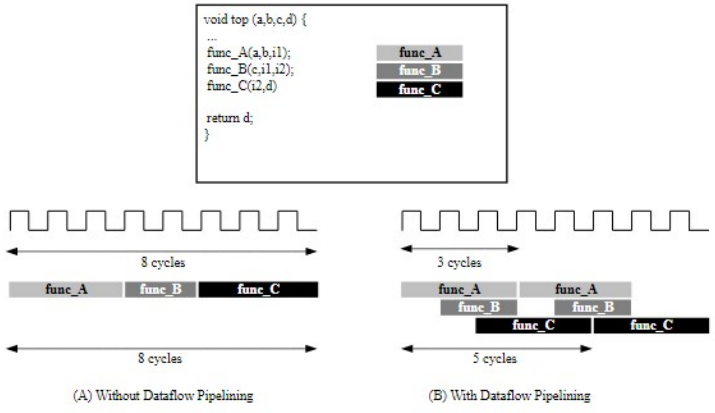

FIGURE 3.4: Applying dataflow directive

For the three sub-functions in this example, the latency and throughput are 8 clock cycles. After applying the dataflow directive, the latency is reduced to 5 clock cycles and throughput to 3 cycles.

However as we have seen in previous sections, there are limitations in applying some directives. For the dataflow directive to work the data must flow through the design from one task to the next. Some coding styles such as bypassing tasks, feedback between tasks, loops with multiple exit conditions may prevent Vivado HLS from performing the dataflow optimization.

3.2.2.6 Pipeline

Syntax:
#pragma HLS pipeline II=<int>

The pipeline directive is one of the most common optimization methods in HLS and it is also used effectively in this project. By using pipeline we allow the concurrent execution of operations ina loop.

The initiation interval(II) of a loop is the number of clock cycles it takes for a loop to process new inputs. When we add the pipeline directive, the II is set to one by default which means the loop will process new data each clock cycle. Otherwise we can specify the II manually.

Here is an example:

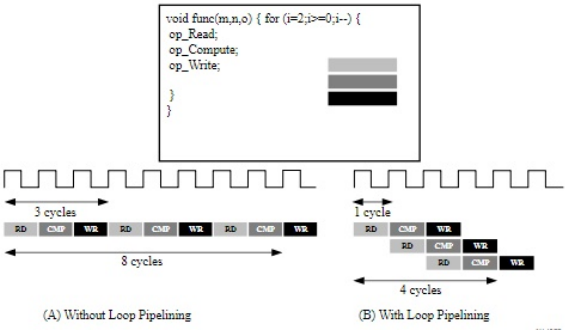

FIGURE 3.5: Pipeline directive

As shown in the picture, for the sequential operation of the loop, 8 clock cycles are required, but with II set to 1, it is reduced to 4 clock cycles.

3.2.2.7 Dependence

Syntax:
#pragma HLS dependence variable=<variable> <class> <type> <direction> distance=<int> <dependent>

Vivado HLS can automatically detect dependencies within loops or between different iterations of a loop. But sometimes the tool is maybe too conservative and does not consider all the false dependencies. In these cases we can manually detect and specify these false dependencies with the dependence directive.
We can specify 2 kinds of dependencies: intra and inter.
Inter: Means the dependency is between different iterations of the same loop.
Intra: Means the dependency is between the same iteration of the loop.

3.2.2.8 Array_partition

Syntax:
#pragma HLS array_partition variable=<name> factor=<int>

In cases where we have large arrays, we can insert this directive to split the array into smaller partitions. The result will be an RTL with multiple smaller memories or multiple registers.

This will of course effect the amount of read and write ports of the memory and requires more memory resources or registers. But the advantage is that using this directive with the UNROLL directive will result in a much better throughput of the design.

3.2.2.9 Array_reshape

Syntax:
#pragma HLS array_reshape variable=<name> <type> factor=<int> dim=<int>

<name>: Indicates the chosen array to be reshaped.
<type>: This field is optional and specifies the type of reshape which can be cyclic, block or complete.

Factor: For example a factor of 2 divides the array in half and doubles the bit-width.
Dim: By indicating this field we specify which dimension of the array we want reshaped.

The array_reshape directive combines array_partition and array_map. Therefor while we have the benifits of array partitioning, by increasing the bit-widths we also reduce the number of block RAMs.

Here is an example of applying array reshape of type "complete":

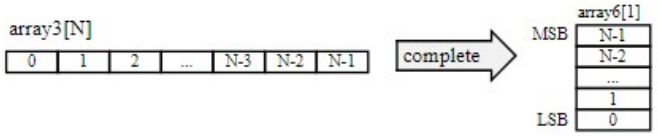

FIGURE 3.6: Array reshape optimization

While applying the array reshape of type "cycle" with factor of 2 will lead to the following optimization:

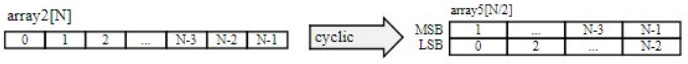

FIGURE 3.7: Cyclic array reshape

3.2.2.10 Interface

Syntax:
#pragma HLS interface <mode> port=<name> bundle=<string>
<mode>: By mode we specify the I/O protocol for the function argument. Vivado HLS supports many protocols but in this project we are using AXI4 interface which is set through m_axi and s_axilite.
Port: which is the name of the function argument
Bundle: which groups the function arguments into AXI interface ports. All function arguments in AXI4-lite and AXI4 are by default grouped into a single AXI4 port.

Since in an RTL design, inputs and output operations must be performed through a port in the design interface, we must use the interface directive in order to specify how RTL ports are created during the interface synthesis.

The Interface directive is used for all kinds of ports in the RTL implementation but in this thesis project we are exclusively using it for the top-level function arguments.

Each function argument can have its own I/O protocol interface such as a valid handshake (ap_vld) or acknoledge handshake (ap_ack). The important thing is that each argument must have a port level interface protocol created for it, including the function return value.

3.2.2.11 Latency Optimization with Loop Tiling

In case of the CNNs, the weights that are used for the training and also the inputs for different layers of the network are too large to be stores in on-chip memory of FPGAs. Therefor external DRAMs must be used to store this data. While it could be a good enough solution for some cases, DRAM accesses are costly in terms of energy and latency and data cashes must be implemented by means of on-chip buffers and local registers.

One solution to solve this problem is the loop tiling technique. In this method the nested loops of the convolutional layer are "tiled". Meaning that the inputs and the weights of each convolutional layer are divided into multiple blocks that can fit in on-chip buffers.

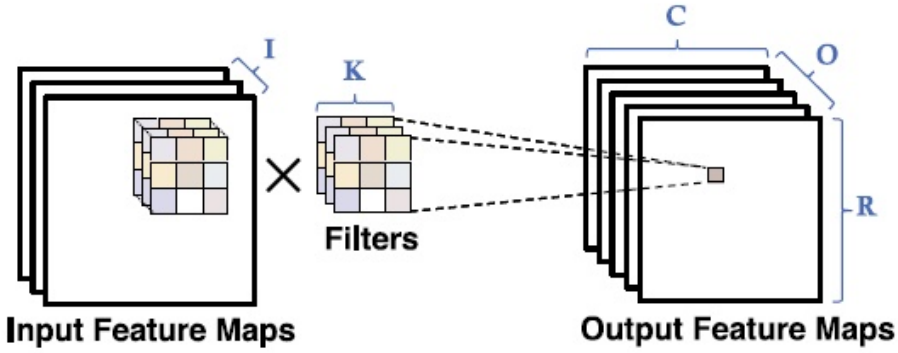

FIGURE 3.8: Tiling

The parallelism to be applied on the code can be implemented in the following method:

While the tile sizes for inputs and the filter can be determined by the designer, the output tile size must be calculated based on the other two:

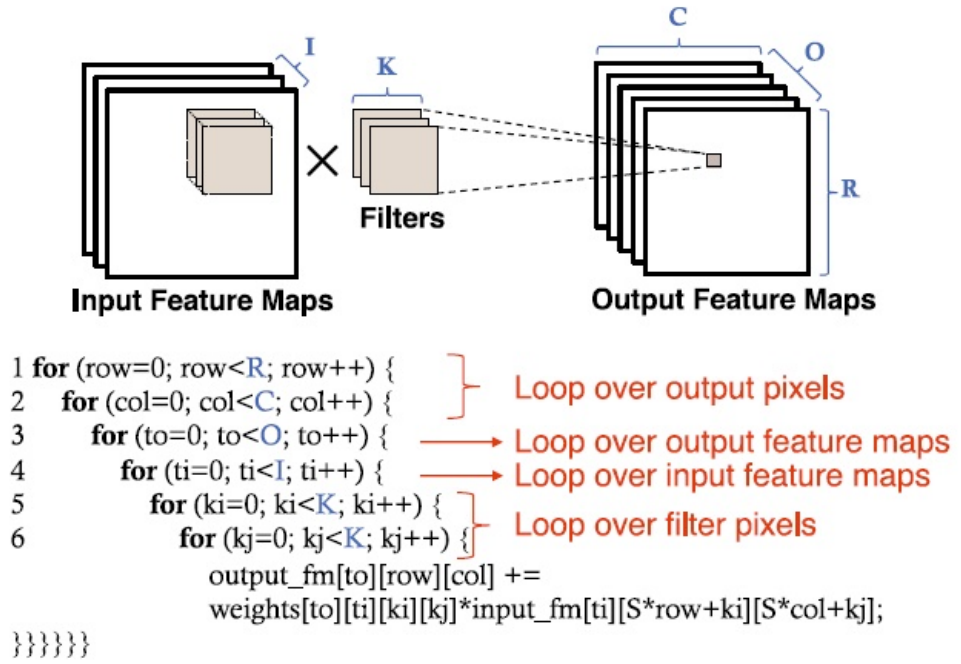

FIGURE 3.9: Tiling implementation

To = (Tr-1) x STRIDE + F

Ti = (Tc-1) x STRIDE + F

Where To and Ti are the height and width of the output tile, Tr and Tc are the height and width of the input tile, and F is the filter size.

3.3 Catapult

3.3.1 Overview of Catapult

Catapult is a tool designed by the Mentor, which can help the designer describe functions and move to an abstraction level.It Has three main parts in sense of high-level synthesis:

1. C/C++/SystemC HLS

The Catapult is the only high-level synthesis platform that supports the high-level description by ANSI C++ and System C, and then based on codes written by an abstract level description language such as C, Catapult can generate optimized Verilog or VHDL, ready for production of RTL synthesis and verification flows.

2.HLS verification

In Catapult, there are three types of verification, the first is checking user's C code before synthesis in order to find errors; the second is verification during simulation, comparing the functionality of user's C source with generated RTL; the third is verifying the code with the RTL from Catapult design checks.

3.Low power HLS

Catapult low power(LP) is a tool that targets power as an optimization goal.The designer can use Catapult LP to explore different hardware architecture and measures the power, performance, and area of each solution.

Chapter 4

HLS and Comparison of the Results

In this chapter, the optimization directives that were described in the previous chapter are applied on YOLO software implementation using two HLS tools: Vivado HLS and Catapult. For the comparison of the results between these two tools, Vivado is used.

4.1 Optimization Techniques

Using the Vivado HLS tool to optimize the YOLO C++ code for hardware implementation, the most used directives are:
Interface
Array_partition
Pipeline
Unroll

4.1.1 Top-level Arguments

As it was mentioned in the previous chapter, in this project we are using the interface directive for the arguments in the top-level function for convolutional layer.
The arguments include:
1. img[H][W][C]: The input image

2. kernel[F][F][C][K]
3. bias[K]
4. out[nH][nW][K]: Output of the conv layer

These arguments are defined not in a single dimension vector to avoid the conversion and using on-chip memory arrays during the convolution process.
Each of these arguments must have a port level protocol created for it.
AXI4 interface is a kind of interfaces that transfers data in sequential streaming method. We are using the AXI4 protocol for each argument through m_axi and s_axilite modes and a seperate bundle for each argument. For example for the input image we will have:

#pragma HLS INTERFACE m_axi port=img offset=slave bundle=gmem3
#pragma HLS INTERFACE s_axilite port=img bundle=control

4.1.2 Conv Layer Nested Loops

We are using the loop tiling technique for the conv layer in this project and as we need to transfer the interface arguments to the tiles on the on chip memory, there are quite a few nested loops in the code.
The first two loops are to transfer the input image and the kernel into the input and kernel tiles respectively.
The pipeline directive is used here directly on the innermost loops because we are working with the function arguments.
For example for transferring the input image pixels you can see the following in the code:

#pragma HLS PIPELINE

input[ii][jj][cc] = img[ii + row][jj + col][cc + ti];

In which "input" is the on-chip tile.

The main convolution process is performed through six nested loops:
The kernel rows and columns, the output tile rows, columns and channels, and the input tile channel.
For the convolution we are using unroll and pipeline directives.
The pipeline is applied on the input tile channel loop. The important issue to notice is

that because of the unroll and pipeline directives applied here, the third dimension of the "input" and "kernel" tiles must be array partitioned as well.

After the main convolution process, another pipeline is applied on the last loop for transfering the final results from the output tile to the "out" function argument.

So far by applying these directives we have managed to reach initiation interval(II) equal to 2. Although this is an acceptable result, we can still optimize the design more.

One technique is to automatic loop pipelining that enables loops to be pipelined automatically based on the iteration count. To do this:
1. Right click on the solution
2. Solution settings
3. General -> Add
4. Command -> config_compile

The pipeline_loops option sets the iteration limit. All loops with an iteration count below this limit are automatically pipelined. The default is 0: no automatic loop pipelining is performed. Therefor if there are loops in the design that we don't want to use automatic pipelining, we can apply the pipeline directive with the off option to that loop. The off option prevents automatic loop pipelining.

The other technique is to take care of the load/store dependencies. If Vivado HLS gives us the warning about memory port limitations in the report, we can change the block RAM with a multi-port RAM. To do this:
1. Right click on the variable in the directive panel
2. Insert directive
3. Choose resource
4. Click on code
5. Choose a RAM with two ports for example RAM-2p-BRAM

Applying these two techniques we now reach the II equal to 1.

4.1.3 Catapult

To perform the optimization in Catapult, the exact same directives are used for the loops including:

#pragma hls_pipeline_init_interval 1

#pragma hls_unroll yes

However there are two main differences between Vivado HLS and Catapult optimizations:

1. The interfaces for the top-level function arguments are applied automatically and implicit in Catapult. So we don't use the "interface" and the "array_partition" directives here.

2. Catapult does not support float data type, so integer data type is used for the function arguments.

4.2 Comparison of the Results

In this section the results between Vivado HLS and Catapult are compared. In order to have a fair comparison we use integer data types for both tools. Also we skip the interface directives in Vivado HLS to have a fair comparison of the utilization resources between the two tools.

We are using Vivado as the tool to compare the Verilog files.
The device chosen for both synthesized files must be the same so a Kintex Ultrascale FPGA is chosen that is available on both tools and the device model is: xcku115-flvf1924-3-e
Also the clock period for both files is set to 10ns.

In Vivado the timing constraints are set the same for both Verilog files. In order to achieve the correct comparison between the latency for the two files, the timing report in Vivado must be checked to control if the user specified timing constraints are met or if the actual clock period achieved in Vivado is different.

Based on this clock period and the number of clock cycles achieved in Vivado HLS and Catapult, the latency is 0.4 ms and 0.5 ms for Vivado HLS and Catapult respectively.

The initiation interval (II) achieved for both files is 1.

You can see the utilization reports obtained in Vivado for both files in the following figures:

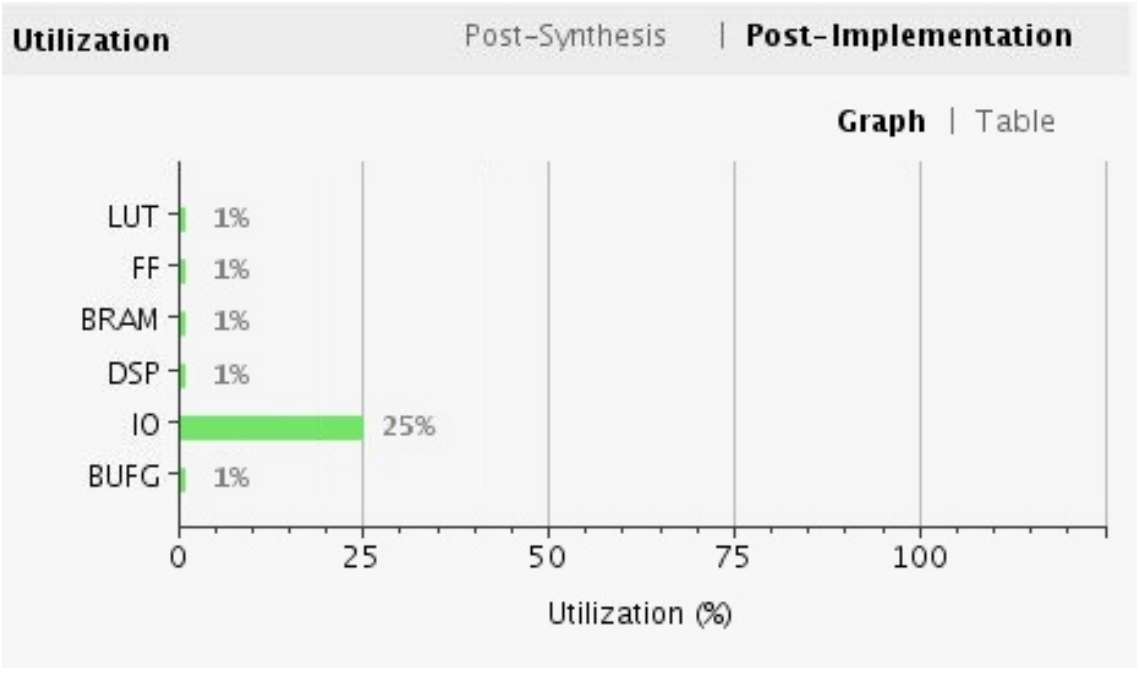

FIGURE 4.1: Utilization graph for Catapult

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	906	663360	0.14
FF	143	1326720	0.01
BRAM	8.50	2160	0.39
DSP	24	5520	0.43
IO	183	728	25.14
BUFG	1	1248	0.08

FIGURE 4.2: Utilization table for Catapult

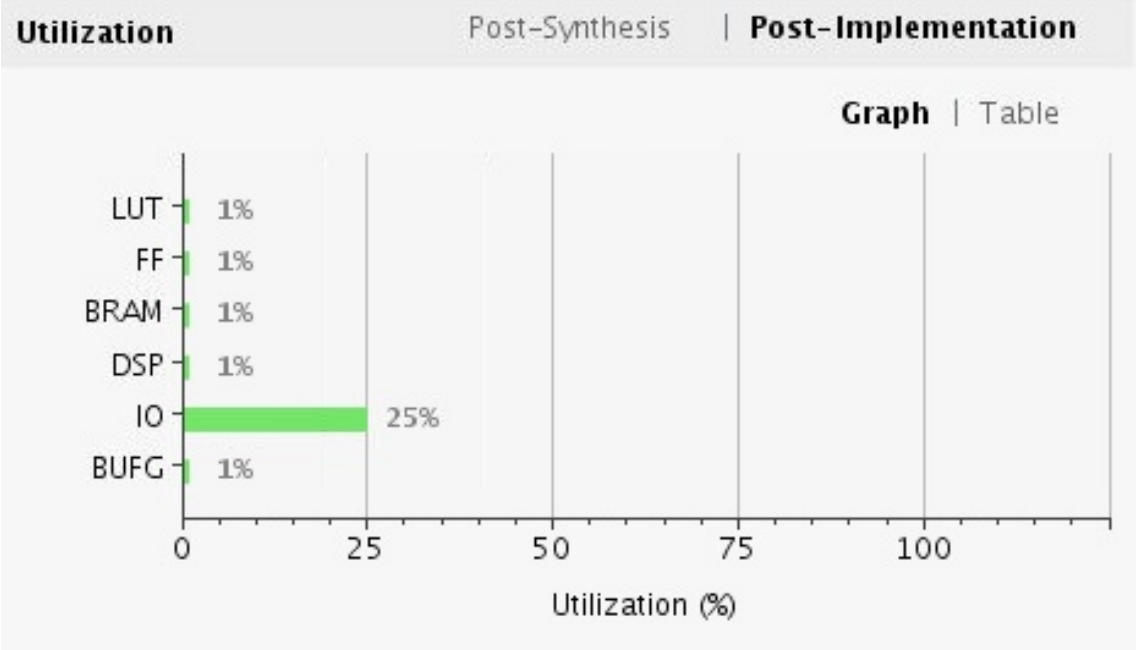

FIGURE 4.3: Utilization graph for Vivado HLS

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	781	663360	0.12
FF	372	1326720	0.03
BRAM	8.50	2160	0.39
DSP	24	5520	0.43
IO	184	728	25.27
BUFG	1	1248	0.08

FIGURE 4.4: Utilization table for Vivado HLS