POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

High-level design of a Depthwise Convolution accelerator and SoC integration using ESP

Supervisors

Prof. Mario Roberto CASU

Dott. Luca URBINATI

Candidate

Riccardo CAPODICASA

A.Y. 2021/2022

Summary

One of the hardest challenge that industry had to front in the last few years was finding a solution to understand the content of an image in a fully automatic way, this branch of study is called "Computer Vision". According to LDV Capital, the number of cameras around the world will proliferate to at least 220% or 45 billion by 2022. This impressive forecast gives one of the reasons why we need techniques to process and classify images in an effective way. From this, one of the most challenging problem in computer vision is "Object Detection", that is the capability to locate object instances inside an image. In order to solve effectively not only the object detection problem, but also an entire set of very complex problems like speech recognition, in 1950s the computer scientist John McCarthy, coined a totally new paradigm called Artificial Intelligence (AI) or "the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do". The goal of this thesis is to realize an hardware accelerator that implements the Depthwise Convolution algorithm, a light-weight convolution algorithm used in Deep Neural Networks targeting mobile applications.

The accelerator is coded in C++, synthesized with High Level Synthesis (HLS) using Catapult HLS and integrated in a System On Chip with a RISC-V processor using ESP (Embedded Scalable Platforms), an open source tool developed by Columbia University. ESP gives the possibility to design accelerators and to integrate them in a SoC, together with processors, memory tiles and input/output interfaces, all connected with a Network On Chip (NoC). After the design phase of the C++ code, we went through the validation, synthesis and simulation steps in order to verify the correct behavior of the Depthwise accelerator. Then we have integrated it into a complete System On Chip (SoC) using the ESP design flow. The realized SoC is composed by our accelerator tile, one memory tile, one I/O tile and one processor tile. In particular, the CPU is a 64-bit Ariane RISC V soft-processor. Finally, after a preliminary simulation and validation phase in Modelsim of the complete SoC, we have implemented it into a real FPGA using a proFPGA xc7v2000t; we have tested our Depthwise Convolution baremetal application on the soft-processor and in particular, we have measured the execution time of the algorithm for both general purpose CPU and dedicated hardware accelerator.

The results have highlighted the differences in terms of speed between our accelerator and the general purpose soft-processor. In fact, using the same convolution parameters, the accelerator takes 15.92 μs to complete the depthwise algorithm, while the CPU takes 231.48 μs. This is an increment in speed of 93.12%. As last step we performed a design-space exploration exploiting the flexibility of HLS to quickly change the accelerator design varying different HLS directives. In particular, we tried to apply different architecture optimizations in order to find a Pareto set of solutions in the Layer Latency vs Area space, spanning from low FPGA resource utilization and high latency (0.0882% of LUTs and 229.1 ms) to high FPGA resource utilization and low latency (0.1722% of LUTs and 139.8 ms).
Thanks to this Design Space Exploration, a hardware designer will easily find the right depthwise accelerator to integrate in her ESP-based SoC that satisfies the overall area and latency constraints.

Acknowledgements

First of all, I want to thank my Family for allowing this to come true. Without them, I would never have been able to get where I am now and be who I am.
I want to thank Sara, who for so many years has supported me in everything I do, and is always at my side in every important step of my life.
I want to thank Davide, Ciro and Nico, who are now like a second family always ready to support me and help me in times of difficulty.
I want to thank Nino and all my classmates, with whom I shared the joys and sorrows of this journey made of many satisfactions but also many curses.
I want to thank all the friends with whom I shared the summers and all my most precious memories.
I want to thank Prof. Casu for giving me the opportunity to work on this interesting project.
I want to thank Luca, who during these months has constantly helped me with evening meetings on Teams long hours and hours trying to understand and solve the most absurd problems.
Last but not least, I want to thank myself for never giving up a single day and being able to achieve this important goal, despite the difficulties.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The object detection problem

One of the hardest challenge that industry had to front in the last few years was finding a solution to understand the content of an image in a fully automated way: this branch of study is called "Computer Vision". According to LDV Capital, the number of cameras around the world will proliferate to at least 220% or 45 billion by 2022 [1]. This impressive forecast gives one of the reasons why we need techniques to process and classify images in an effective way. One of the most challenging problem in computer vision is "Object Detection", that is the capability to locate object instances inside an image. Essentially, given an input image, the goal of the algorithm is to determine if there are instances of specific objects or not in it, and if it finds the desired targets, to return also the spatial location of those instances. We could make an endless list of possible applications on this topic, for example a video-surveillance system that automatically recognize the license plate of a car that is making a traffic infraction [2] or an algorithm that with a better accuracy than traditional techniques is capable to diagnose cancer just from an histopathological image [3]. These examples help us to understand that object detection is an important problem for society. Object detection is divided in two different approaches: the first is the recognition of a broad set of generic object categories, the second is the recognition of specific object categories. We can see some examples in Fig. 1.1. Once the object is detected, its spatial location is often highlighted with a rectangular bounding box (the most used in practice), a pixel-wise segmentation mask or a closed boundary [4] as reported in Fig. 1.2 (a), (b) and (c) respectively.

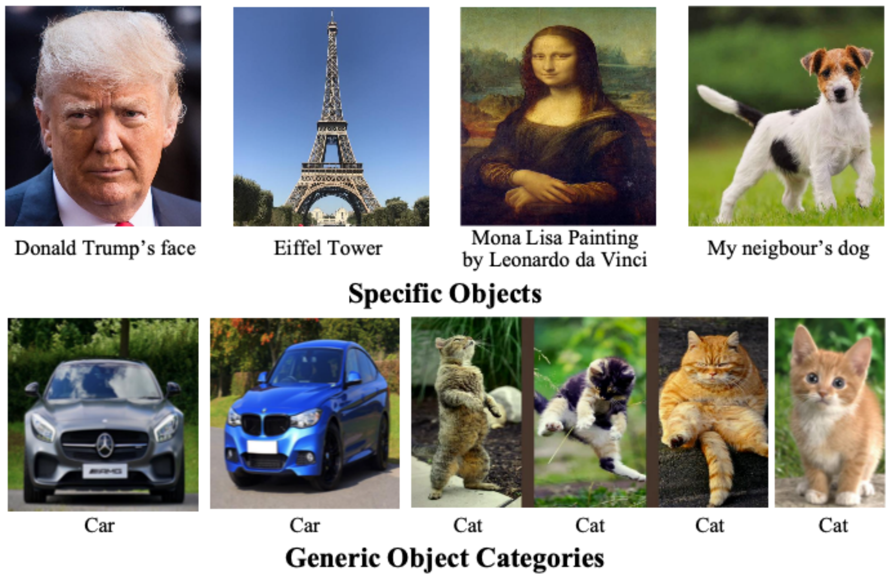


Figure 1.1: Generic and Specific Objects (Image taken from [4])

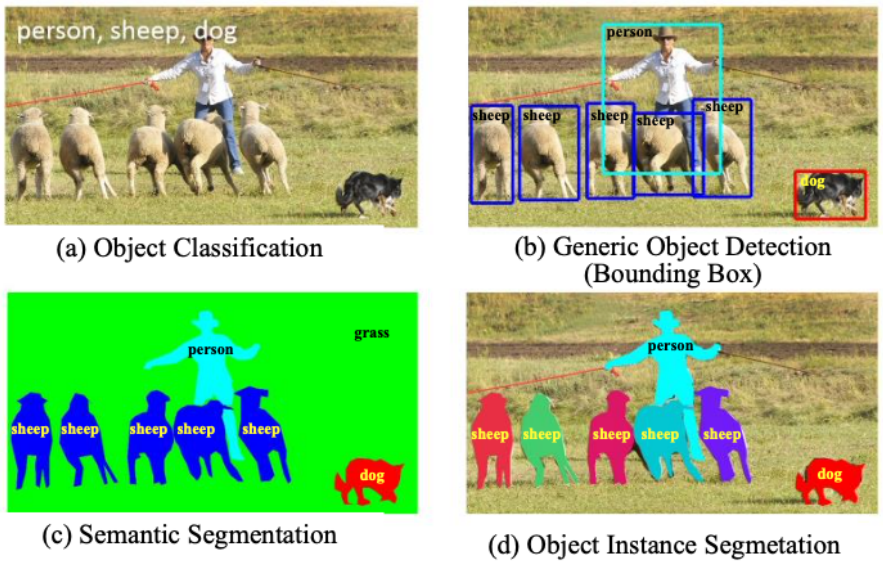


Figure 1.2: Types of objects detection with different boundaries around objects (Image taken from [4])

Why is this technique so challenging? First of all, we have to find and classify all the object instances inside our image from a pre-determined set of object classes (Object Classification) in Fig 1.2 (a). Then in case we want to recognize a dog, we know that there are a lot of different dogs in the world, with different shapes, colors, sizes, and most importantly, we can have the same dog that appears differently in multiple images. In fact it can have different orientations, levels of zoom, poses or light exposures. So our classifier needs to take care of all of these options in order to recognize a dog in every possible scenario. We have also the spatial localization requirement, so we need to check and understand where each object is inside the image. Some examples of what discussed above are in Fig. 1.3.

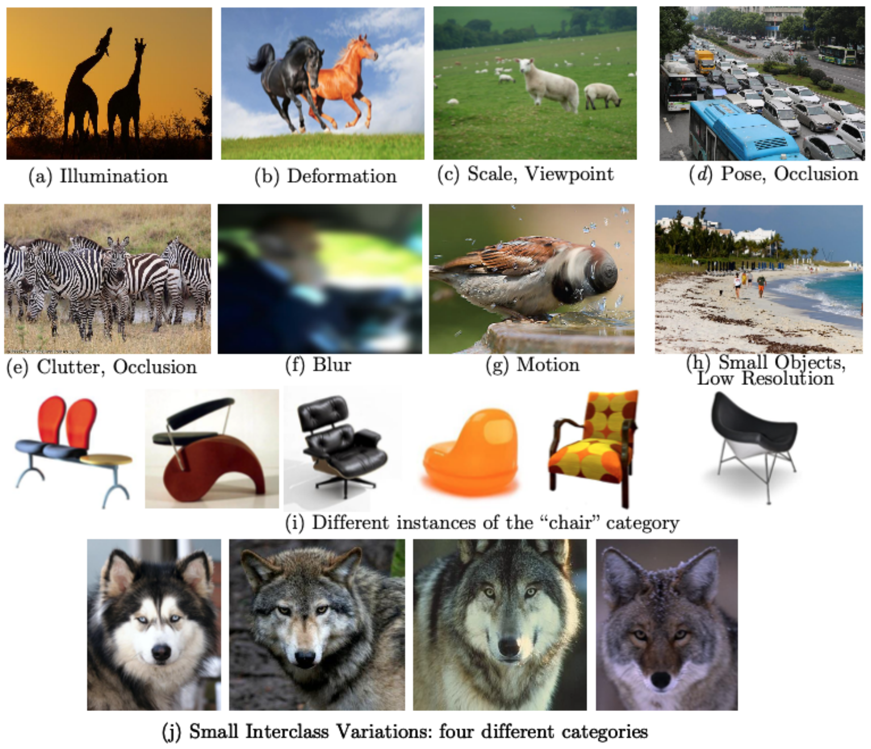

(a) Illumination
(b) Deformation
(c) Scale, Viewpoint
(d) Pose, Occlusion
(e) Clutter, Occlusion
(f) Blur
(g) Motion
(h) Small Objects, Low Resolution
(i) Different instances of the "chair" category
(j) Small Interclass Variations: four different categories

Figure 1.3: Various challenging conditions for object detection (Image taken from [4])

We can identify two different strategies to solve the object detection problem: high accuracy detection and power efficient detection [4]. The optimal decision is a trade-off between these two approaches in order to find the best solution for the specific application. Now that we are aware of the complexity behind a correct detection of an object, we need an accurate and efficient model in order to implement this algorithm.

1.2 Artificial Intelligence: a new paradigm

In order to solve effectively not only the object detection problem, but also an entire set of very complex problems like speech recognition, a totally new paradigm is necessary: Artificial Intelligence (AI). It "the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do", according to McCarty's definition dated back to 1950s[5]. So the AI concept is not new but only starting from 1990s it started to be applied to solve real problems [6], because of the exponential growth of computer performances, lower cost and grater capacity of memories and last but not least the huge amount of data available on internet. Inside AI we can find many other hierarchical sub-fields, as we can see in Fig. 1.4.

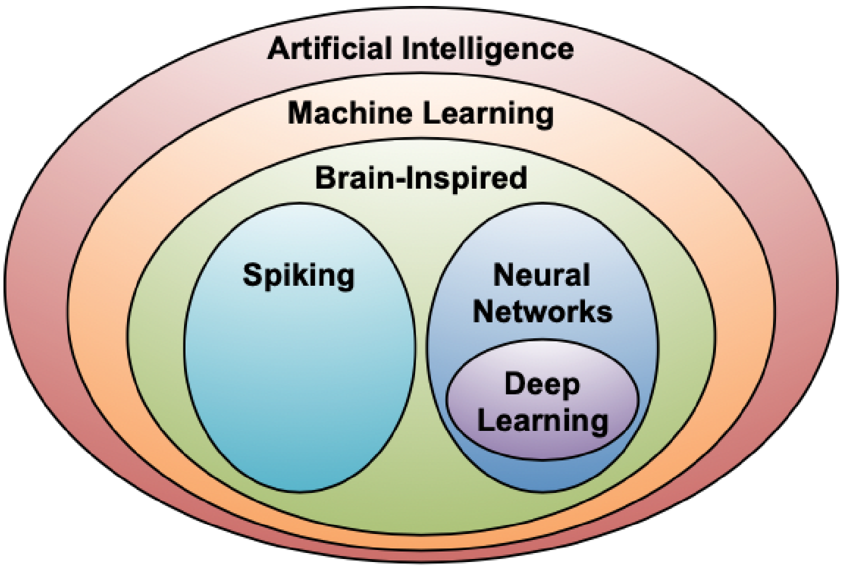


Figure 1.4: The hierarchical sub-fields of AI (Image taken from [5])

1. The first layer is Machine Learning or "the field of study that gives computers the ability to learn without being explicitly programmed" as Arthur Samuel defined in 1959 [5]. This is a revolutionary approach because in this way we can write programs that will be able to automatically learn things and modify themselves dynamically in real time during their execution. The main limitation of this method is that these types of algorithms can handle effectively only specific tasks and we need to perform a preliminary training procedure in order to set their parameters needed for a correct execution.

2. The second layer that is immediately inside the machine learning field is the so called Brain-Inspired computation. This layer is focused on implementing algorithms trying to emulate the human brain that indeed is the best working machine capable of solving almost every problem by learning [5]. The approach of these algorithms is to replicate some features that are strictly related to the working principle of the human brain. In order to understand how this is done, we need to introduce how the brain is supposed to work accordingly to the last researches. First of all, we need to know that the main computational elements of the brain are called neurons. All neurons are connected each other through two different links: the first one is the dendrite, that enters the neuron, while the second is the axon, that leaves the neuron. Essentially, one neuron takes a signal that comes from a dendrite as input, performs some computations and then provides an output that exits from an axon. These input and output signals are also called "activations" [5] and the connection between an axon and other dendrites is called synapse.

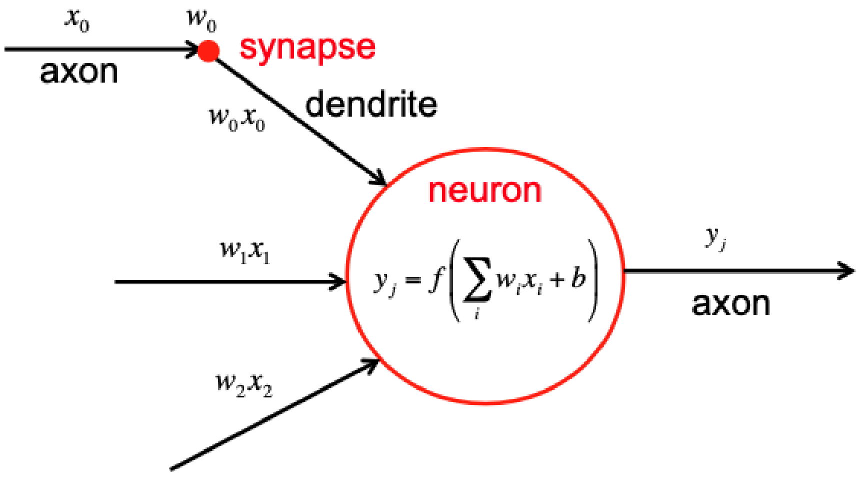


Figure 1.5: Artificial neuron structure (Image taken from [5])

The peculiar thing of the synapse is that it scales its input signals x_i by weight factors w_i as we can see from Fig. 1.5. We need to underline that nowadays the real behavior of our brain is still unknown and this method is only an interpretation of what we know so far about it. Furthermore, we are not able to completely reproduce this behavior and in order to do that we need to introduce some simplifications that make AI still far from a real human brain.

3. In the next layer, inside the Brain-Inspired computation, we have an area that is called Spiking Computing. It derives from the fact that, in reality, the

signals that come in the synapses are electrical pulses and the information that will be read by the neuron is not only related to the spike's amplitude, but also to its arrival time.

4. Inside Brain-Inspired there is also another important area called Neural Networks that in turns contains "Deep Learning". These two sets will be the focus of the next chapter.

1.3 Thesis focus

The goal of this thesis is to realize an hardware accelerator that implements the Depthwise Convolution algorithm, a light-weight convolution algorithm used in Deep Neural Networks targeting mobile applications.
The accelerator is coded in C++, synthesized with High Level Synthesis (HLS) using Catapult HLS and integrated in a System On Chip with a RISC-V processor using ESP (Embedded Scalable Platforms), an open source tool developed by Columbia University. ESP gives the possibility to design accelerators and to integrate them in a SoC, together with processors, memory tiles and input/output interfaces, all connected with a Network On Chip (NoC). After the design phase of the C++ code, we went through the validation, synthesis and simulation steps in order to verify the correct behavior of the Depthwise accelerator. Then we have integrated it into a complete System On Chip (SoC) using the ESP design flow. The realized SoC is composed by our accelerator tile, one memory tile, one I/O tile and one processor tile. In particular, the CPU is a 64-bit Ariane RISC V soft-processor. Finally, after a preliminary simulation and validation phase in Modelsim of the complete SoC, we have implemented it into a real FPGA using a proFPGA xc7v2000t; we have tested our Depthwise Convolution baremetal application on the soft-processor and in particular, we have measured the execution time of the algorithm for both general purpose CPU and dedicated hardware accelerator. The results have highlighted the differences in terms of speed between our accelerator and the general purpose soft-processor. In fact, using the same convolution parameters, the accelerator takes 15.92 μs to complete the depthwise algorithm, while the CPU takes 231.48 μs. This is an increment in speed of 93.12%. As last step we performed a design-space exploration exploiting the flexibility of HLS to quickly change the accelerator design varying different HLS directives. In particular, we tried to apply different architecture optimizations in order to find a Pareto set of solutions in the Layer Latency vs Area space, spanning from low FPGA resource utilization and high latency (0.0882% of LUTs and 229.1 ms) to high FPGA resource utilization and low latency (0.1722% of LUTs and 139.8 ms). Thanks to this Design Space Exploration, a hardware designer will easily find the right depthwise accelerator to integrate in her ESP-based SoC that satisfies the overall area and latency constraints.

1.4 Thesis outline

The reminder of this thesis is organized as follows:

- **Chapter 2:** introduces Neural Networks and Deep Neural Networks focusing on Convolutional Neural Networks, providing a small description of all the main layers of this type of networks. In particular the main focus is on their mathematical formulations. It deals especially with the standard 2D-convolution and the depthwise convolution algorithms and their comparison in terms of reduction of operations and parameters.
- **Chapter 3:** explains more in detail what is ESP and how it works, starting from all the flows that the user can use in order to design her own project and how ESP implements all the blocks that are already available and that are automatically inserted when creating an SoC.
- **Chapter 4:** shows how to design an accelerator in ESP and how to design an accelerator using Catapult HLS in ESP. First, it explains the directory tree of ESP. Then, it focuses on the main file and folders describing how to modify each file and each sample source code provided by ESP in a tutorial-like way in order to make very easy to replicate the steps of this thesis. Finally it explains the behavior of the accelerator showing the simulation and validation results obtained in Questasim. So this manuscript wants to contribute to improve the poor documentation regarding this particular design flow Catapult HLS+ESP.
- **Chapter 5:** follows the same approach of the previous chapter. Initially it shows how to create a custom SoC using the ESP GUI. Them it focuses on how to modify the application code in order to make it work with the Catapult HLS design flow. It concludes by showing shows how to program and test our design on the target FPGA providing the simulation results obtained in Modelsim.
- **Chapter 6:** shows how we can perform the design space exploration in Catapult HLS. Then it discusses the effects of the tuned design parameters and HLS directives on the latency and area choosing different optimization techniques.
- **Chapter 7:** it reports the relevant results that comes out after all the tests that are executed on hardware and outlines some possible future works associated to this project.

Chapter 2

Neural Networks and DNNs

2.1 What is a Neural Network?

Resuming the concepts from the previous chapter, we have seen what AI is and, in particular, we have analyzed all the sub-fields related to this topic, going from Machine Learning to Artificial Neural Networks (ANNs) passing through Brain-Inspired computing. The idea that a neuron works by adding the weighted sum of the input values is the source of inspiration for ANN. In particular, as we have seen in the previous chapter, these weights are applied directly to the input axons in order to provide in output a scaled version of their inputs. However a neuron does not provide in output only this simple weighted sum but a non-linear version of it. The non-linearity is introduced by a non-linear function called "activation function" [5]. To keep the one-to-one correspondence between the biological brain model and the artificial one, the standard approach is to apply some non-linear operations to this scaled sum. Also in ANNs we do not have only a single operation but we will have a sort of cascade of different neurons all connected by different synapses. In practice, what happens in the simplest case of a three layer ANN is that the neurons that are in the "input layer" perform an operation on their inputs, then they provide a particular set of outputs that are be propagated through the synapses in a middle layer. This middle layer often called "Hidden Layer" repeats the previous weighted sum operation followed by a non-linearity on its inputs and through other synapses propagates its outputs to the "output layer". Finally, the "output layer" provides the final output of the entire ANNs. Often in literature the output of the neurons are also called "activations" while the synapses are called "weights". we can see this network in Fig. 2.1 [5].

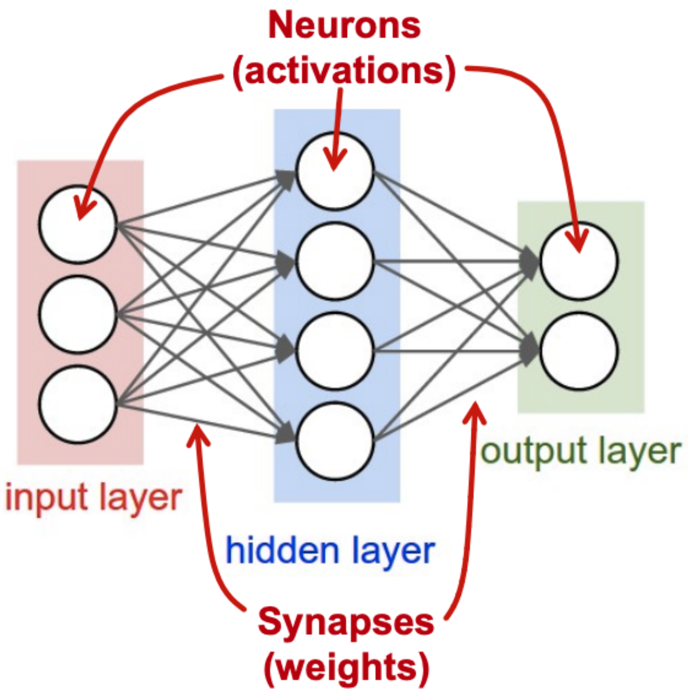


Figure 2.1: Neural Network with a single hidden layer (Image taken from [5])

A mathematical formulation that describes how the hidden layer is obtained from the input layer is:

$$Y_j = f(\sum_{i=1}^{3} W_{ij} \cdot X_i + b_j) \tag{2.1}$$

We can visually see this equation in Fig. 2.2.

W_{ij} are the weights, X_i are the input activations, Y_j are the output activations, b_j is a simple bias term of the j-th output neuron and f is the non-linear function of the considered layer. As we said before, this is the simplest possible configuration, in fact inside the Neural Networks field there is another area called "Deep Learning" recalling Fig. 1.4. Within the category of Deep Learning, ANNs become "Deep Neural Networks" (DNNs). These networks have more than three layers that means more than one single hidden layer and they can reach also thousands of layers [5]. The great advantage of this approach is that using a DNN we can assign to each layer the capability to identify a particular feature from the input. The result is that combining the features extracted by each layer the network becomes more powerful and achieves better performance for a given task [5]. If we translate this into the Object Detection problem, we can have for example a DNN where the pixels of an image are given in input to the first layer.

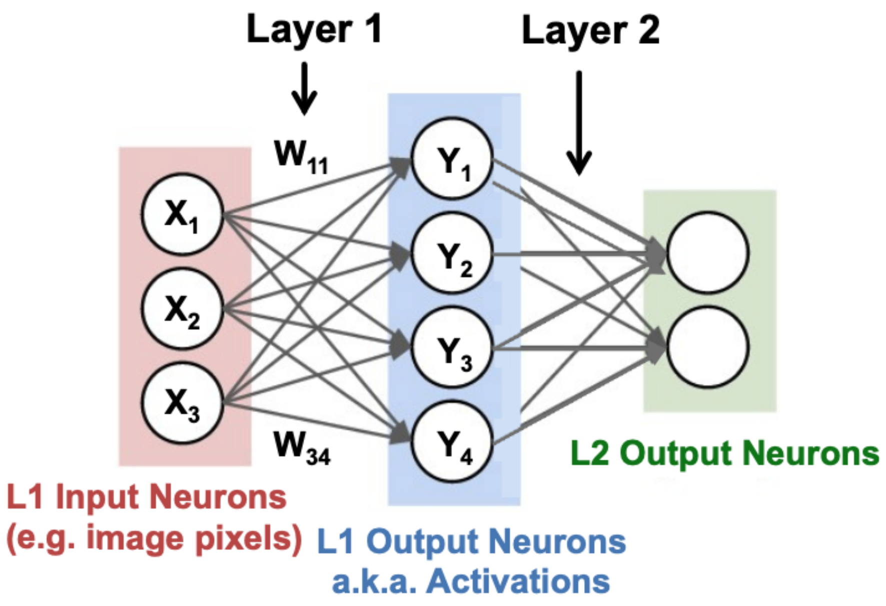


Figure 2.2: Computation of the intermediate output (Image taken from [5])

The output of this initial layer could be for example the interpretation of some low-level features like lines and edges. Then the next layer will take the output features of the previous layer as inputs and will modify them in order to identify more complex (high-level) features like shapes. After that we will have another layer combining again all the resulting features to identify something more and more complex. Finally with all these information derived from the combined features, the network provides a probability that the initial input corresponding to these features belongs to a specific object class.

2.1.1 Neural Networks learning process

The learning process, also known as "Training", is performed by modifying the values of the weights (and eventually also of the bias) of the network [5]. In order to test the performances of our program, we need to run it after the training process and check the validity of the outputs with respect to the known inputs. This is called "Inference" process[5]. If we refer to the Object Classification problem, we can say that when we use a DNN we provide in input the pixels of our image and at the output we will have a vector of probabilities that the object inside the image belongs to a known class. The network will predict the object in the image to be part of the class with the higher output probability or score like in Fig. 2.3.

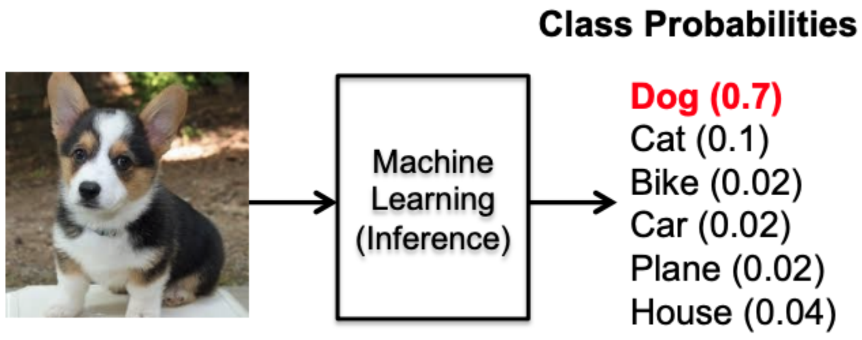


Figure 2.3: Inference example (Image taken from [5])

The goal of the training process is to adjust the weights of the network to maximize the probability of a correct class prediction and in the meantime to minimize the probability of an incorrect one. In the ideal case we would have a DNN that has a 1.0 probability corresponding to the correct class and a probability of 0.0 corresponding to all the incorrect classes.

The gradient descent technique

Considering the case of "Supervised Learning", where each training sample is associated to a "label", that is the correct/true class it belongs to, during training we are interested in the difference[1] between the true class score and the one computed by our DNN with the current set of weights: this difference is called "Loss" (L) [5]. So the actual goal of the training process is to find a specific set of weights that minimizes this loss on the input data set that we are processing. This data set is also called "training set". The most used technique to do this operation is called "Gradient Descent". In short a multiple of the gradient of the loss relative to each weight (w_{ij}) is used to update the weight itself at the end of each "epoch", that is the time that a network takes to process all the samples in the training set. The equation that describes this process for an epoch is:

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha \cdot \frac{\partial L}{\partial w_{ij}} \tag{2.2}$$

where α is called "Learning Rate" [5], t is the epoch, i is used to iterate on the data inside a layer and j is used to iterate among the layers. So as we can see from the equation this is an iterative process and it tells us how the weights change at each

[1]In reality there are a lot of Loss Functions and in general they can do a lot of different operations not only the simple difference

epoch. Usually, in order to efficiently do this gradient operation, a technique called "Backpropagation" is used. We can visually see this operation in Fig. 2.4 [5]. Since the focus of the thesis is not on training, but on inference of DNNs on hardware accelerators, we suggest resource [5] to the interested reader.

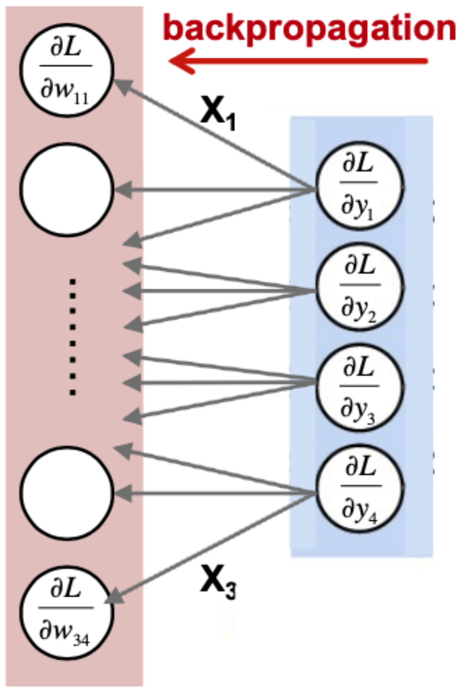

backpropagation
$\frac{\partial L}{\partial w_{11}}$
x_1
$\frac{\partial L}{\partial y_1}$
$\frac{\partial L}{\partial y_2}$
$\frac{\partial L}{\partial y_3}$
$\frac{\partial L}{\partial y_4}$
x_3
$\frac{\partial L}{\partial w_{34}}$

Figure 2.4: Backpropgataion (Image taken from [5])

2.1.2 Different types of DNNs

Depending on the application, we can choose between different types of Deep Neural Networks, where each one is specifically made to be particularly efficient in solving specific problems. In particular, a first distinction that we can make is related to how the network computes the output starting from the input and we can identify two possible solutions:

- **Feed-forward networks:** in this type of networks, each output is computed using as inputs the outputs coming from the previous layer. Following this idea, the final outputs of the network are simply obtained using as inputs the outputs of the second to last layer. In this Deep Neural Networks we do not need to memorize any information because the outputs of a certain layer are always independent on the outputs obtained in a previous computation in the same layer [5].

- **Recurrent neural networks:** on the contrary, in this type of networks we need an internal memory in order to store some intermediate outputs that will be used as inputs for the subsequent computations. In other words, we are computing new outputs using new inputs and previous outputs from the same layer [5].

We can see these two different structures in Fig. 2.5

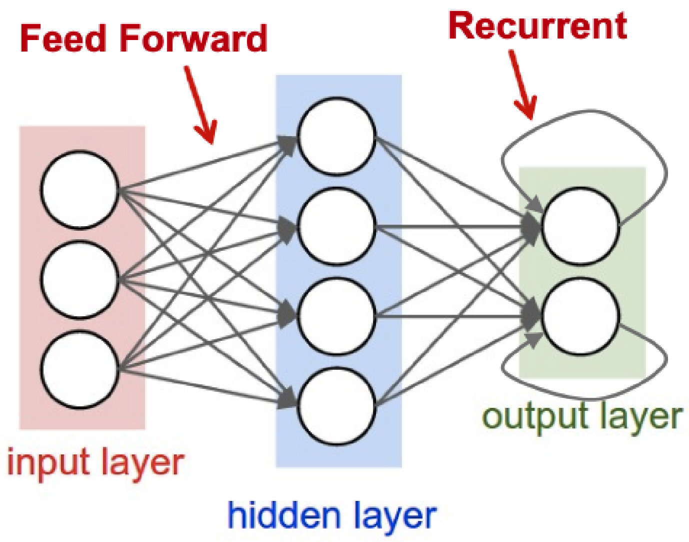


Figure 2.5: Feed-forward and Recurrent NNs (Image taken from [5])

A second distinction is instead related to how the weights are connected from a layer to the next one:

- **Fully-connected layers:** In the Fully-connected layer (FC) all the layer's outputs are connected to all the next layer's inputs. Essentially for each output neuron of a certain layer we make the weighted sum using all the neurons of the previous layer as inputs. Since each couple of neurons between two layers requires a unique weight the total number of weights of a FC network is huge, as well as the required memory to store them and the number of multiplications between inputs and weights to perform each inference. If all the layers of a DNN are made in this way, this type of networks is also called "multi-layer perceptrons (MLP)"

- **Sparsely-connected layers:** Differently from the previous case, with sparsely-connected layers we can remove some connections between the outputs of a layer and the inputs of the next layer simply forcing the relative weights to zero [5].
- **Weight sharing:** There is also a third approach that is called weight sharing. In this case if an output is only dependent on a specific set of inputs or on a fixed window of inputs, we can physically delete the connection with the other inputs. Then we can share every time the weights that belong to a specific window in order to compute in a very efficient way the relative output [5].

In Fig. 2.6 we can see the differences between the Fully-connected and Sparsely-connected layers.

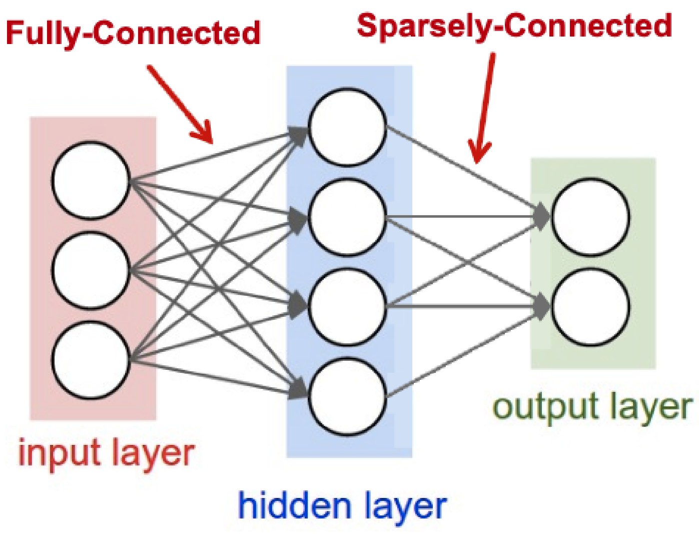
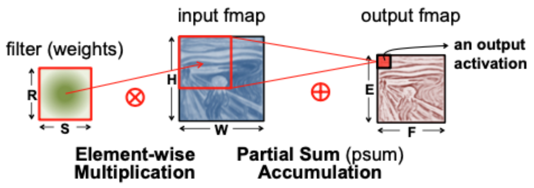


Figure 2.6: Fully-connected and Sparsely-connected layers (Image taken from [5])

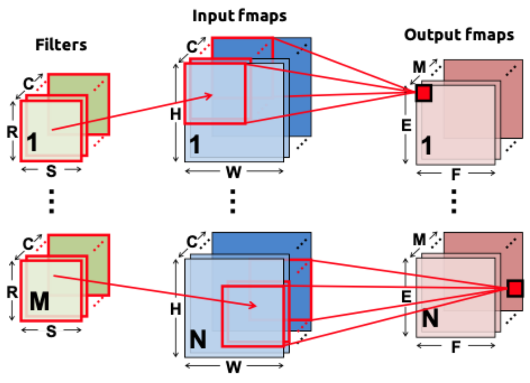
One of the most common windowed and weight-shared layer is the so called Convolutional layer or CONV layer that, as suggested by the name, does a convolution operation between inputs and weights inheriting all the features from the sparsely-connected layers and weight-shared layers.

2.2 Convolutional Neural Networks

In this section we analyze in detail a particular Deep Neural Network called Convolutional Neural Network or simply CNN. This is one of the most used Neural Network in Object detection problems and its peculiarity is the use of a multiple set of convolutional layers in order to determine its final output. Each layer provides to the next one an higher level of abstraction of the input, also called "Feature Map" or simply "fmap" that, layer-by-layer, extract and retains the most important information of that specific layer of the network [5]. A convolutional layer takes a tensor as input (i.e. a three-dimensional matrix) that could be seen as a stack of feature maps (two-dimensional matrices called "Channels") along the third dimension. Then it applies a convolution operation with another tensor that could be seen as a stack of weights (two-dimensional matrices called "kernels") along the third dimension. it does an element-wise multiplication between a channel and the specific associated 2D kernel, then it sums each partial result obtaining one value of the output feature map (ofmap), in this way it will have in output one single channel. One possibility is to convolve multiple 3D kernels with the same input tensor in order to have in output a multi channel ofmap. We can see this operation in Fig. 2.7.



(a) 2-D convolution in traditional image processing



(b) High dimensional convolutions in CNNs

Figure 2.7: 2D and high dimensional convolution in CNNs (Image taken from [5])

Another possible operation is using a multiple set of ifmaps called "Batches" by using for all of them the same filter tensor. We can also try to write in a formal way the specific operation that is done by a convolutional layer. First of all we need to define all the parameters that are needed:

- N = Batch size, with index z
- M = number of ofmap channels, with index u
- C = number of ifmap channels
- H/W = ifmap height and width
- R/S = filters height and width
- E/F = ofmap height and width, with indexes y and x

In particular we set:

- $0 \leq z < N$
- $0 \leq u < M$
- $0 \leq x < F$
- $0 \leq y < E$
- $E = (H - R + 2P)/U$
- $F = (W - S + 2P)/U$

Where we have:

- $\mathbf{O}$ = ofmaps matrices
- $\mathbf{I}$ = ifmaps matrices
- $\mathbf{W}$ = filters matrices
- $\mathbf{B}$ = biases matrices
- U = stride value
- P = Padding value (zero padding applied to each side)

Now we can write our equation [5]:

$$\mathbf{O}[z][u][x][y] = \mathbf{B}[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} \mathbf{I}[z][k][Ux+i][Uy+j] \cdot \mathbf{W}[u][k][i][j] \tag{2.3}$$

Nowadays a modern Convolutional Neural Network is composed by multiple layers, not only CONV layers. First of all, if we want to perform Object classification we need a Fully Connected layer at the end of the network simply because after the feature extraction we need to classify the results in order to correctly detect the target object between all the possible classes. Not only, in fact a large set of optional layers has been used in order to improve the entire network performances. Now we analyze the most used in the order in which they are usually stacked:

1. **Non Linearity Layer:** After each convolutional layer it is usually inserted a layer in order to introduce a non linearity function directly on the new feature map. In the literature have been proposed many non linear functions. One of the most used function in the past was the sigmoid function or the hyperbolic tangent. Nowadays, due to its good ratio between network accuracy and implementation simplicity, the rectified linear unit (ReLU), defined as $y = max(0, x)$ has become very popular [5], with all its variations like leaky ReLU or exponential ReLU, we can summarize all these functions in Fig. 2.8.

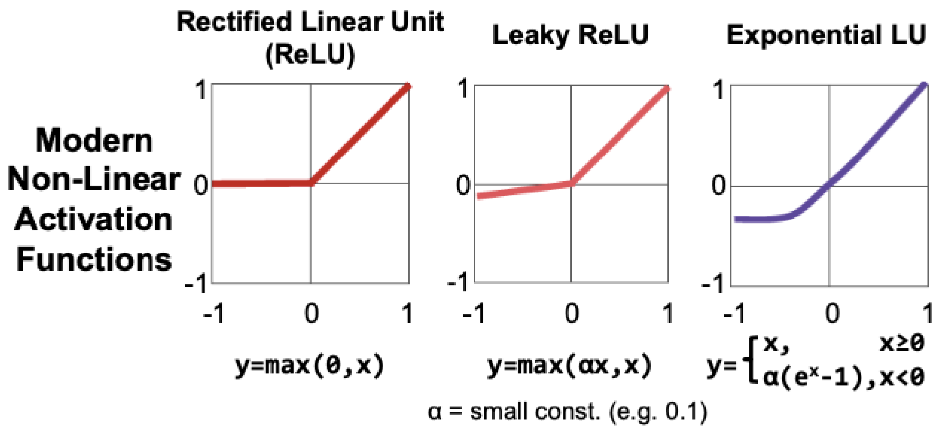


Figure 2.8: ReLu functions (Image taken from [5])

2. **Pooling Layer:** Then we can optionally insert the so called Pooling layer that reduces the feature maps size, keeping the number of channels constant. This

pooling computation is applied to each channel separately and has the big advantage to give to the network the capability to be insensitive to any possible small noises (for example small shifting of our values) [5]. This operation is done dividing the original feature map into equal sub matrices. For example if we have a 4x4 feature map, we can divide it into four 2x2 little matrices. After that, we can apply two different techniques:

(a) **Max Pooling:** The new ofmap is made by the maximum numbers of each sub matrix.

(b) **Average Pooling:** The new ofmap is constituted by the average of each sub matrix.

In Fig. 2.9 we can see these pooling operations.

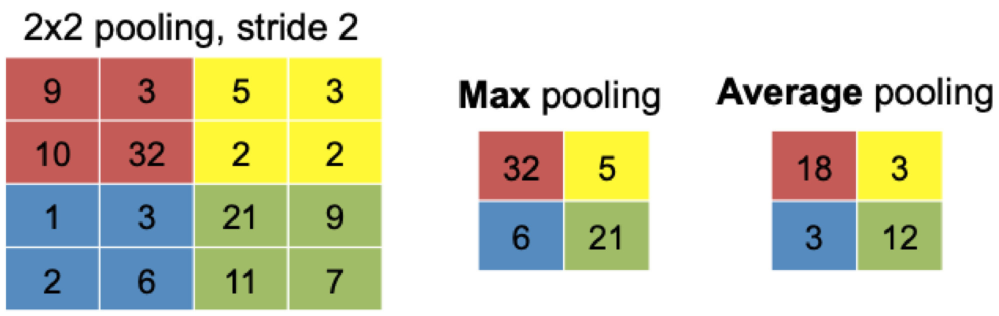


Figure 2.9: Different types of Pooling layer (Image taken from [5])

3. **Normalization Layer:** A very useful operation that is applied to input data is to normalize inputs in order to control their distribution and then accelerate the training process improving also the overall accuracy. After that, in modern networks, the "Batch Normalization" (BN) technique is applied, in practice the previous normalized value is further scaled and shifted using three additional parameters γ, β and ϵ. γ and β are learned from training while ϵ is a constant used to avoid numerical problems [5]. We can subdivide this operation in four steps:

   (a) At each batch all the mean and standard deviation values are computed for every feature map for every "x,y" position in the input matrix.

   (b) Then each feature map is normalized for the corresponding values of mean and standard deviation in order to have a zero mean ($\mu = 0$) and a unit standard deviation ($\sigma = 1$).

(c) After that the Batch normalization is applied following this equation:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta \tag{2.4}$$

(d) Finally this operation is repeated for each training batch.

After the training process all the parameters: mean, standard deviation, γ, β and ϵ will be fixed and they will not change during the inference process. It is important to notice that this normalization layer+BN is usually applied between the convolutional layer and the non linear layer.

2.3 Energy Efficient Convolutions

Object detection nowadays is not the only challenge that we have to face. In fact, the rise of mobile devices and their widespread diffusion introduce the problem of power efficiency. This is because we want to use Machine Learning algorithms without draining the available battery and at the same time keeping all the advantages related to their very high performances. What we can do is defining new efficient ways to compute the convolution operation in order to build some energy efficient CONV Layers.

2.3.1 Depthwise Convolution

Depthwise Convolution is one of the most popular algorithm used to bring Image Classification capabilities on mobile devices with very high energy efficiency compared to the classical Convolution algorithm. In Depthwise convolution we apply each kernel to each ifmap. For example, if we have a filter with 3 channels and an image with 3 channels, what it does is convolving the corresponding image with corresponding channel and then stacking them back to create the final output feature map [7] as shown in Fig. 2.10.

A standard 2D convolution takes a D_F x D_F x M feature map tensor $\mathbf{F}$ as input and produces a D_O x D_O x N feature map tensor $\mathbf{G}$ where D_F is the spatial width and height of a square input feature map, D_O is the spatial width and height of a square output feature map, M is the number of input channels and N is the number of output channels [8]. The weight tensor $\mathbf{K}$ of a standard convolution is D_K x D_K x M x N, where D_K is the width and height of a square kernel. Using these parameters, we know that the ofmap, assuming a stride of one and no padding is computed as [8]:

$$\mathbf{G}_{k,l,n} = \sum_{i,j,m} \mathbf{K}_{i,j,m,n} \cdot \mathbf{F}_{k+i-1,l+j-1,m} \tag{2.5}$$

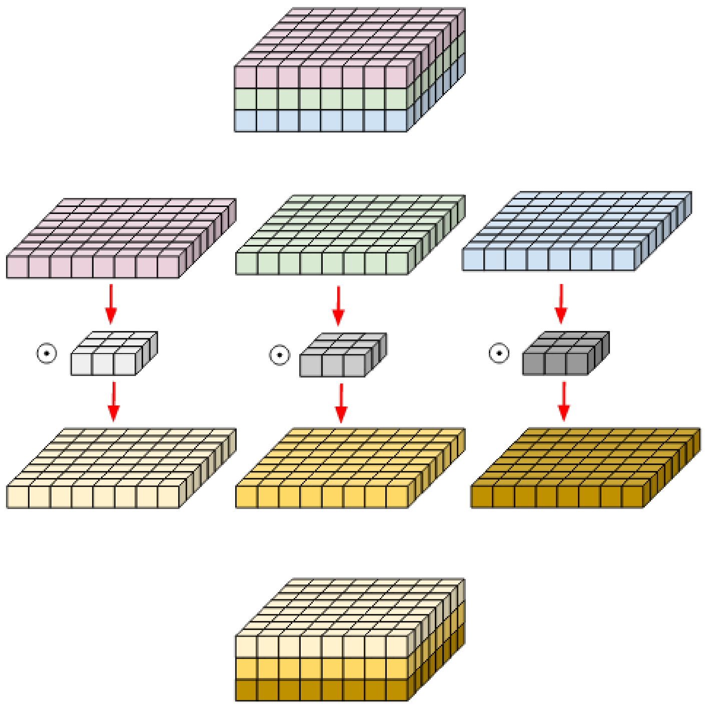


Figure 2.10: Depthwise Convolution (Image taken from [7])

and from this we can derive the MAC operations number for the algorithm that will be:

$$C_{convolution} = D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{2.6}$$

As we can see from the equation above, the computational cost of a 2D convolution depends linearly on the number of input channels, output channels, feature map size and kernel size. This is because with the standard convolution we filter and combine at the same time the ifmap into the ofmap, so the computational cost depends linearly both on the kernel size (responsible for filtering) and on the number of output channels (responsible for combining). With the Depthwise convolution we do not perform the channel-wise sum of the output features, instead we stack them together into a multi-channel ofmap. In order to see the difference between the Depthwise and the standard convolution MAC operations number, we can write down the equation that tells us how the ofmap is computed using the Depthwise Convolution:

$$\mathbf{G'}_{k,l,m} = \sum_{i,j} \mathbf{K'}_{i,j,m} \cdot \mathbf{F'}_{k+i-1,l+j-1,m} \tag{2.7}$$

Now **K'** is the depthwise convolutional kernel that has a size of D_K x D_K x M . Differently from the standard convolution, in the depthwise operation we have that the m-th filter in the tensor **K'** is applied directly to the m-th channel in the input feature map **F'** producing the m-th channel of the output feature map **G'** [8]. Now we derive the MAC operations number for the Depthwise Convolution [8].

$$C_{depthwise} = D_K \cdot D_K \cdot M \cdot D_F \cdot D_F \tag{2.8}$$

We can notice how the Depthwise Convolution is way more efficient compared with the standard convolution because we lose the linear dependency on the number of output channels N. Given a CNN, a 2D convolution layer cannot be directly substituted with a Depthwise Convolution because we will lose the channel-wise addition of the output features. For this purpose we use the so called Depthwise Separable Convolution.

Depthwise Separable Convolution

The Depthwise Separable Convolution is an algorithm that separate a standard convolution into a Depthwise Convolution followed by a 1 x 1 2D convolution, called Pointwise Convolution. We can see this algorithm in the example showed in Fig. 2.11

So, while a standard convolution filters and combines at the same time the ifmap into the ofmap, the Depthwise Separable Convolution splits these operations in two different "sub-operations", one for filtering and one for combining. This is exactly like dividing the original CONV Layer in two different layers, one for each sub-operation. This separation drastically reduces the computational cost of the algorithm. Recalling the computational cost of the Depthwise Convolution of 2.8 and adding the contribution of the Pointwise Convolution we get the total computational cost of the Depthwise Separable Convolution [8]:

$$C_{depthwise\ separable} = D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F \tag{2.9}$$

Finally, we can perform the comparison between the Depthwise Separable Convolution and the standard 2D convolution by simply computing the ratio between the MAC operations number:

$$C_{reduction} = \frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2} \tag{2.10}$$

This is really a not negligible difference.

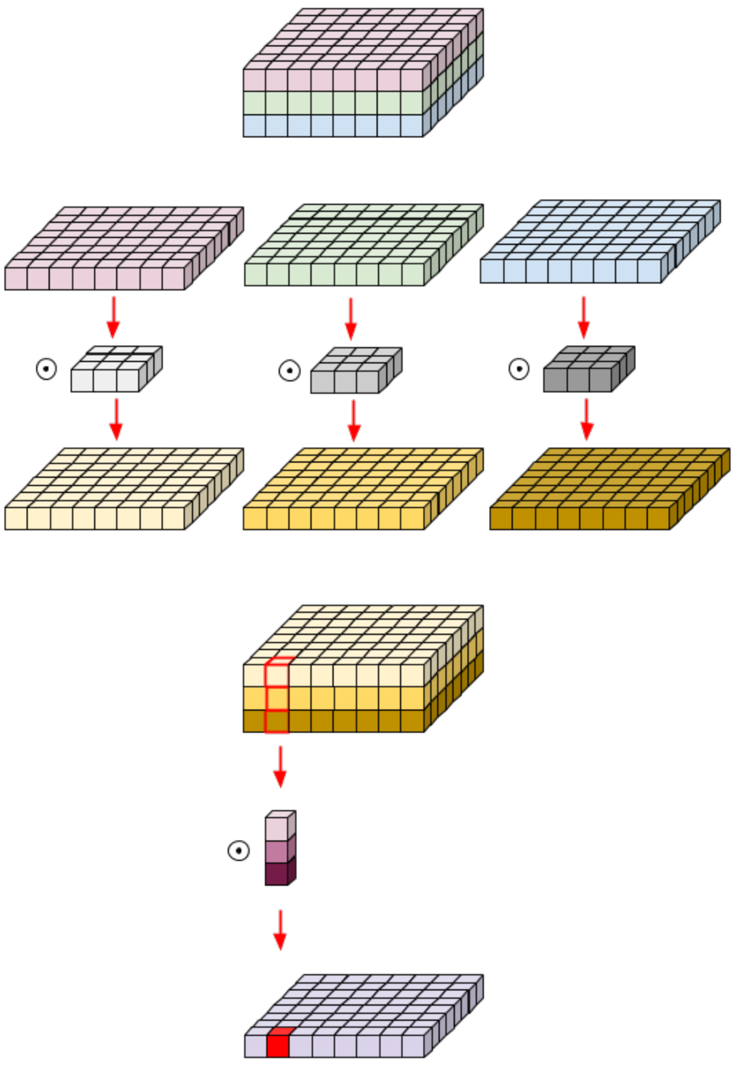


Figure 2.11: Depthwise Separable Convolution (Image taken from [7])

Chapter 3

Embedded Scalable Platforms (ESP)

3.1 What is ESP

Embedded Scalable Platforms (ESP) is an open-source research platform for heterogeneous SoC design and programming [9]. The main reason behind the development of this tool is the exponential growth of heterogeneous computing. In fact, nowadays, the majority of computing systems rely on highly heterogeneous SoC architectures [9]. These architectures are called heterogeneous because they embed general-purpose processors like CPUs, special-purpose processors like graphics processing units, and very specialized accelerators, like image processing or speech recognition ones, in the same SoC. This transition from homogeneous multi-core processors to heterogeneous SoCs is driven by the need to obtain very high energy-efficient computation [9]. Within ESP several design and integration flows for SoCs are provided. This makes very easy to design complex and large SoCs that could be implemented into FPGAs for testing or prototyping. With ESP is also possible to design and implement custom accelerators using one of the following supported design flows, also reported in Fig. 3.1 [9]:

- C/C++ with Xilinx Vivado HLS
- C/C++ with Mentor Catapult HLS
- SystemC with Cadence Stratus
- Keras, PyTorch, ONNX and TensorFlow with hls4ml [10]
- Chisel, SystemVerilog and VHDL for RTL design

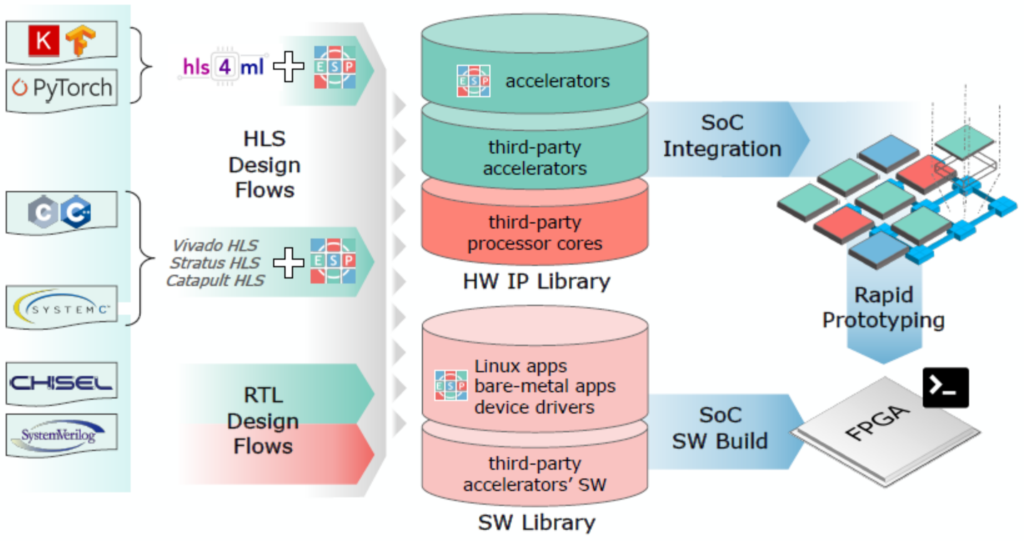


Figure 3.1: ESP design and integration flow (image taken from [9])

ESP provides also an intuitive and interactive GUI in which the entire SoC is seen like a simple tile grid and the components that we want to introduce in the SoC are the tiles that fill this grid. The user can choose the structure of the grid selecting how many tiles and which kind of tiles have to include in the SoC. For example, if we choose a grid dimension of 3 x 3, we are able to create a SoC with 9 different tiles inside, like shown in Fig. 3.2.

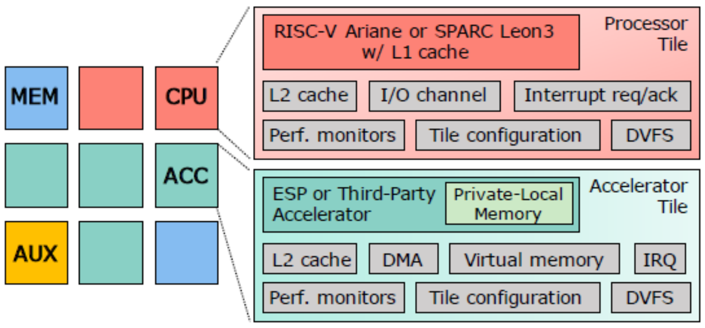


Figure 3.2: Example of a 3 x 3 grid structure in ESP (image taken from [9])

In ESP it is possible to choose between four different kind of tiles:

- **Processor tile:** it implements a CPU in the SoC;
- **Accelerator tile:** it an hardware accelerator in the SoC;
- **Memory tile:** it implements the communication mechanism between the SoC and the main memory;
- **Auxiliary tile:** it implements the connection with the external peripherals.

All these tiles are interconnected with a complex multi-plane Network On Chip (NoC) [9]. This is one of the main features of the ESP architecture because with this approach it has a modular structure where each tile can be designed independently from the others simplifying the entire design process. This is a completely new and different approach. The standard approach focuses only on the processor, designing the rest of the system depending on it. On the other hand with this modularity, ESP moves the focus to the entire system where all its tiles are on the same level of importance. [9].

3.1.1 NoC Architecture

The NoC is an intermediate layer used for interconnecting all the tiles in a complete transparent way from the user point of view. This transparency is ensured by the fact that all the tiles have specific components and interfaces called "sockets" in order to be completely decoupled from the NoC interface. For example in Fig. 3.3 we can see the detailed architecture of a six-plane NoC where the modularity provided by ESP with all the available tiles mentioned before is highlighted. The description of this complex interconnection system will not be addressed by this thesis because it is out of its scope but it is possible to read all the details in document [9].
Now we will provide more details about all the different tiles available in ESP.

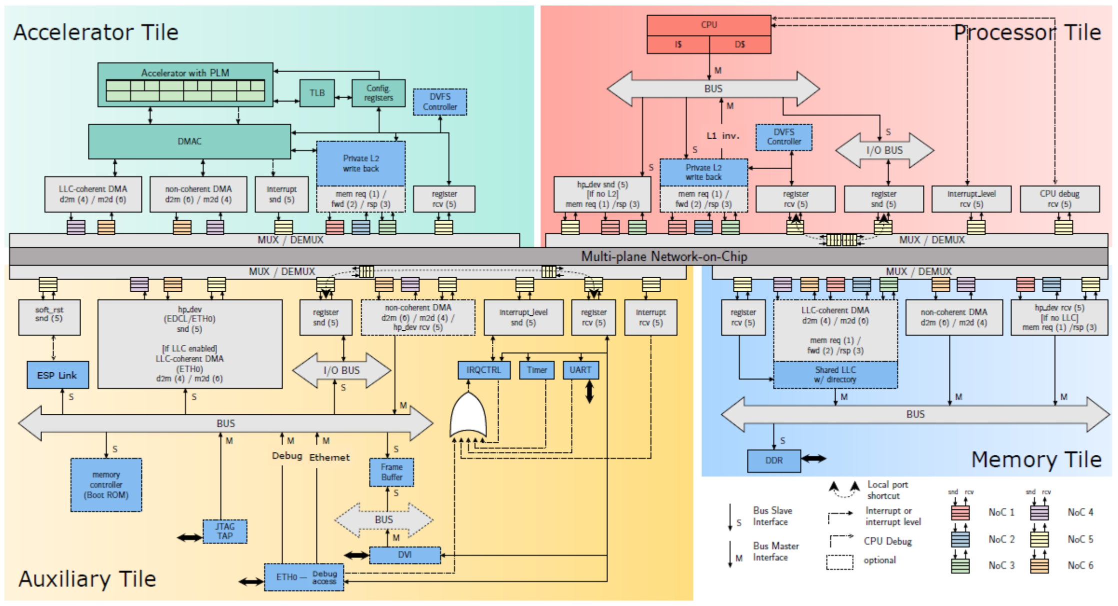


Figure 3.3: Detailed NoC architecture and interconnection with the different tiles (image taken from [9])

3.1.2 Processor Tile

This tile allows to choose during the design phase between two different CPUs: the RISC-V Ariane core from ETH Zurich [11, 12] and the SPARC 32-bit LEON3 core from Cobham Gaisler [13]. They provide a private L1 cache layer and both can run baremetal applications and/or Linux operating system. This ESP tile provides also a size configurable unified private L2 cache layer that implements a directory-based MESI cache-coherence protocol [9]. Thanks to the NoC CPUs do not need extra communication layers because each processor tile has an internal local bus communication system. In particular, the LEON3 processor requires a 32-bit AHB bus interface, while the Ariane processor a 64-bit AXI interface [9]. All the IO operations required by the CPU are directly forwarded to the NoC plane using an APB adapter. The only custom communication protocol available for this tile is the one that implements the protocol between the CPU, the interrupt controller and the system timer available in the auxiliary tile.

3.1.3 Memory Tile

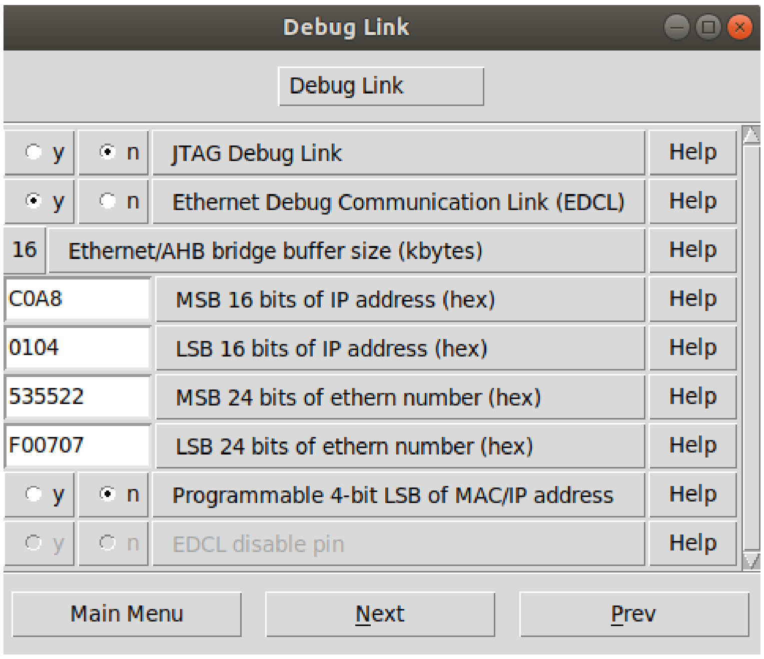
This tile is composed by a channel to an external DRAM [9]. In particular the designer can choose the number of memories that the SoC should have. When a memory tile is placed in the SoC, all the logic that manages the different partitioning of the overall memory is automatically inferred by ESP. Inside the memory it is also present a configurable partition that is used for implementing the Last Level Cache (LLC). The LLC works together with the CPU L2 cache described before using the MESI protocol [9]. This cache structure is used to support an operating system in a Symmetric Multi-Processor configuration allowing also the configuration of a coherent structure for the accelerators [14].

3.1.4 Accelerator Tile

This tile implements the architecture of a loosely-coupled accelerator [15]. This specific architecture is able to perform all its tasks in a completely independent way from the CPU and it exchanges all the data directly with the memory. In ESP the accelerator is modeled using simple and common interfaces: load/store ports, in order to exchange data with the memory tiles; configuration signals, used to provide the required parameters to the accelerator including the start and done signals. The latter is used to send an interrupt to the CPU when the accelerator has finished its algorithm. This structure is automatically realized by the ESP design flow. In addition, ESP provides an accelerator design flow that allows to integrate also third-party accelerators. A particular run-time configuration that is really important for an accelerator is the one related to the Coherence protocol service. This feature makes the SoC extremely flexible because it allows to have in a single SoC a perfect integration of all the implemented heterogeneous accelerators [9]. The non-coherent DMA architecture allows an accelerator to communicate with the memory tiles without passing through the cache hierarchy. Instead, the fully-coherent approach allows the communication between the accelerator and an optional private cache installed in the accelerator tile. The LLC-coherent DMA and the coherent DMA models are supported by the ESP cache hierarchy as an addition to the directory-based MESI protocol, in which accelerators submit requests directly to the LLC without having a private cache. While the coherent DMA maintain the accelerator requests consistent with regard to all of the system private caches, the LLC-coherent DMA does not. The ESP cache hierarchy handles the coherent DMA and fully-coherent DMA approaches completely in hardware, while the non-coherent DMA and the LLC-coherent DMA need some additional synchronization mechanisms implemented in software using specific HLS directives. All these mechanisms are not explored in this thesis, but more details can be found in [9].

3.1.5 Auxiliary Tile

This tile contains all the peripherals used by the SoC to communicate with the external world: an UART interface, a debug link to monitor ESP SoCs on FPGA discussed later, a digital video interface, the Ethernet NIC, and a monitor module capable of sending performance information via Ethernet. As we have seen from Fig. 3.3, this tile is the most complex one because, depending on the SoC structure it has to provide all the connections required by the accelerators to the external world and to the other tiles. We can find for example: an interrupt level proxy, that is meant to handle the communication between the interrupt controller and the CPU; an Ethernet proxy, that allows to access an ESP SoC via SSH protocol remotely; a frame-buffer memory directly connected to the memory-mapped I/O proxy, that can be used by accelerators and processors to write to the video output for debugging purposes. In addition, ESP provides a dedicated application called *ESP Link*. It allows to debug the system connecting to it remotely via Ethernet. In case of a profpga FPGA, this is done through a dedicated Ethernet debug interface, that is a board that has to be mounted on the profpga FPGA [9]. ESP Link has a very simple GUI showed in Fig. 3.4.



y [✓ n]	JTAG Debug Link	Help
[✓ y] n	Ethernet Debug Communication Link (EDCL)	Help
16	Ethernet/AHB bridge buffer size (kbytes)	Help
C0A8	MSB 16 bits of IP address (hex)	Help
0104	LSB 16 bits of IP address (hex)	Help
535522	MSB 24 bits of ethern number (hex)	Help
F00707	LSB 24 bits of ethern number (hex)	Help
y [✓ n]	Programmable 4-bit LSB of MAC/IP address	Help
y n	EDCL disable pin	Help

Figure 3.4: ESP Debug Link application (image taken from [16])

As we can see, it is possible to configure the static IP of the debug unit on the FPGA side by simply changing the IP address fields in order to make the connection with the host. In particular, given an ESP SoC on FPGA, there are two possible ways to access it through Ethernet from a host machine: with a direct link or through a router. We will not provide other details because we did not use this tool, but a better explanation is present in the ESP documentation [16]. Instead, the other peripherals like timer, interrupt controller, bootrom and UART are controlled directly by all the master devices, that in ESP are the other tiles, thanks to the proxy pairs embedded in the auxiliary tile (one proxy for the master and the other for the slave).

Chapter 4

Design an accelerator using the ESP flow

The ESP accelerator design flow allows the creation and then the insertion of a custom accelerator into the ESP accelerators library. In this way every-time we create a new accelerator it can be automatically instantiated and used in a SoC following the dedicated design flow. As we already discussed in the previous chapter, it is possible to use different languages and different abstraction levels in order to make our accelerator. We can do for example a cycle-accurate RTL description or an un-timed behavioral HLS description, but it is also possible to use hls4ml in order to directly synthesize an accelerator starting from deep learning models like PyTorch or TensorFlow/Keras [9].

4.1 Design an accelerator using HLS

ESP provides several ESP-compatible accelerator templates together with HLS-ready skeletons that, in addition to the documentation, simplifies a lot the entire design process for the HLS flow. There are mainly three reasons behind this choice to put a lot of effort supporting the HLS design flow [9]:

1. There are a lot of already existing algorithms written in C/C++ that could be used with HLS to design an accelerator;
2. The use of HLS simplifies the co-design between hardware and software because we can use the same testbench written in C language to validate both; moreover the testbench can be used as baremetal application for tests on hardware
3. With HLS we can immediately do a functional verification simply running our code, while using a classical hardware design we need to do a complete

RTL simulation that takes much more time. On the other hand, HLS becomes inefficient if we want to describe complex and detailed designs that have particular timing, architecture, and communication constraints.

4.1.1 Accelerator interfaces and internal structure

As we said in the previous chapter, the ESP accelerator tile is a loosely-coupled architecture [15]. It exchanges big data sets with the memory hierarchy and carries out coarse-grained computations. Fig. 4.1 depicts the architecture and user interface of a generic ESP accelerator.

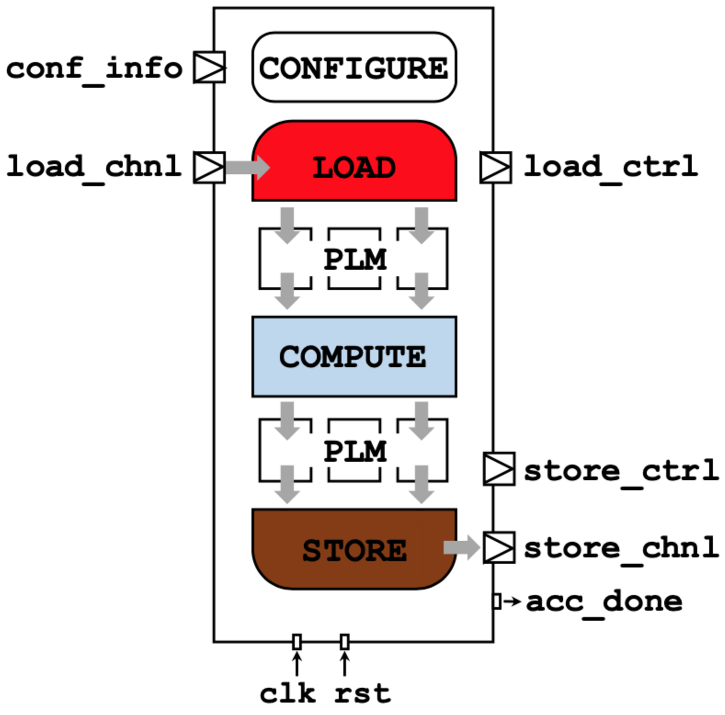


Figure 4.1: Common structure of an ESP accelerator tile (image taken from [9])

In particular with these interfaces the accelerator can perform:

1. The communication with the processor tile using the memory-mapped registers via the *conf_info* channel.
2. The configuration of the DMA controller using the *load_ctrl* and *store_ctrl* channels.

3. The actual data exchange with the DMA controller using the *load_chnl* and *store_chnl* channels.
4. The notification sending mechanism when it finishes the algorithm using the *acc_done* signal.

HLS implements all these channels using latency-insensitive primitives. Both in the communication within the accelerator and in the communication across the NoC, these primitives maintain functional correctness in the presence of latency fluctuation. In particular, this is performed using *ready* and *valid* signals. The valid signal notifies that the data packet in the channel in the current clock cycle is valid, while the ready signal tells if a component is processing or not, for example the accelerator de-asserts it when it performs a memory access. Another thing that we can notice from Fig. 4.1 is that ESP schedules the accelerator execution in four different phases:

1. **Configure:** In this phase an external software application (for example the testbench) configures the parameters, and provides the *start signal* to the accelerator using the memory-mapped registers.
2. **Load:** Here the accelerator using the DMA reads the data from the main memory tile, and stores it into a private local memory (PLM) inside the accelerator itself.
3. **Compute:** The accelerator performs the actual algorithm, in our case it is the Depthwise Convolution and writes the results into a dedicated PLM
4. **Store:** In this last phase the accelerator writes the results into the main memory tile from its output PLM using again the DMA mechanism.

It is important to underline that these PLMs are completely customizable by the designer, in fact she can choose the banks organization and the amount of ports available on them.

4.1.2 Available templates and automatic code script

ESP provides also several accelerator templates spread across all the supported HLS flows. These templates already embed the accelerator description with the interfaces and the internal structure discussed in the previous section. In particular, this could be extremely useful for example to understand all the coding guidelines and all the HLS directives that are used in order to not infer unwanted memories in the design minimizing the overall area. An additional tool offered by ESP for accelerator design is an interactive script that creates an accelerator skeleton that is fully functional and HLS-ready from a set of parameters supplied by the designer.

This skeleton contains simple templates and placeholder functions that the designer could manually customize in order to make its own application specific accelerator. The parameters required by the script are: unique name and ID, desired HLS tool flow, a list of application-specific configuration registers, bit-width of the data tokens, size of the data set and number of batches of data sets. Then, after generating the skeleton, the designer has to customize the computing algorithm in the corresponding computation phase of the accelerator, the generation of the inputs and also the validation of the outputs inside the testbench and in the baremetal software application and finally organize the PLM structure.

4.2 Catapult HLS design flow

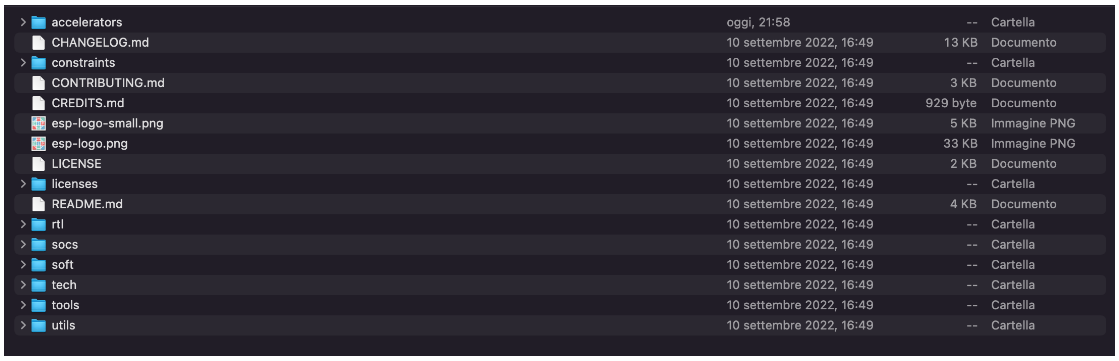
The HLS design flow that we have chosen for this thesis is the C++ Catapult HLS design flow. This is because right now, it is the only flow that supports the C co-simulation in ESP. On the other hand, during the time of this thesis, it does not support the script for the automatic generation of the code and the only support provided by the ESP documentation is a sample accelerator that performs the Softmax algorithm [17] with a tutorial step by step on how to implement it in a SoC and then in a FPGA after the initial validation phase. For this reason, we have written this chapter as a more detailed tutorial that can be integrated in the ESP documentation. In particular, our goal is to realize a Depthwise accelerator with these maximum parameters:

- **Maximum ifmap dimension:** 16 x 16
- **Maximum kernel dimension:** 5 x 5
- **Maximum number of channels:** 16
- **Maximum stride:** 2
- **Maximum padding:** 6 (3 for each side)

So we focused our attention on how to modify the Softmax example in order to realize this accelerator, but the adopted strategy can be followed also for a generic accelerator with minor modifications.

4.2.1 Files & Directory hierarchy

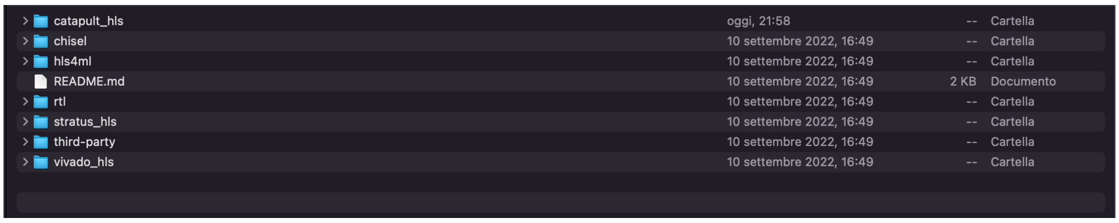
First of all, we need to introduce how ESP is organized in terms of files and directory hierarchy. In Fig. 4.2 we can see what is inside the *esp* root folder.



```
accelerators          oggi, 21:58                     --  Cartella
CHANGELOG.md          10 settembre 2022, 16:49     13 KB  Documento
constraints           10 settembre 2022, 16:49        --  Cartella
CONTRIBUTING.md       10 settembre 2022, 16:49      3 KB  Documento
CREDITS.md            10 settembre 2022, 16:49  929 byte  Documento
esp-logo-small.png    10 settembre 2022, 16:49      5 KB  Immagine PNG
esp-logo.png          10 settembre 2022, 16:49     33 KB  Immagine PNG
LICENSE               10 settembre 2022, 16:49      2 KB  Documento
licenses              10 settembre 2022, 16:49        --  Cartella
README.md             10 settembre 2022, 16:49      4 KB  Documento
rtl                   10 settembre 2022, 16:49        --  Cartella
socs                  10 settembre 2022, 16:49        --  Cartella
soft                  10 settembre 2022, 16:49        --  Cartella
tech                  10 settembre 2022, 16:49        --  Cartella
tools                 10 settembre 2022, 16:49        --  Cartella
utils                 10 settembre 2022, 16:49        --  Cartella
```

Figure 4.2: *esp* root folder

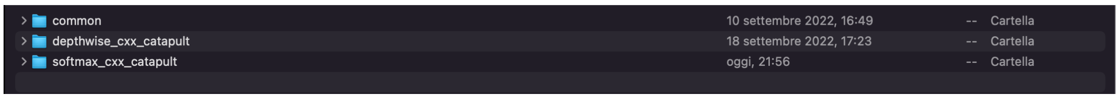
In particular, we need to focus on the tree of directories and files that regard the accelerators. It is very useful to understand all the files that we have to modify in order to build or debug our accelerator are located. So we need to move inside the *accelerators* folder. Here we can see that there are several folders, one for each different accelerator design flow, as shown in Fig. 4.3.



```
catapult_hls          oggi, 21:58                     --  Cartella
chisel                10 settembre 2022, 16:49        --  Cartella
hls4ml                10 settembre 2022, 16:49        --  Cartella
README.md             10 settembre 2022, 16:49      2 KB  Documento
rtl                   10 settembre 2022, 16:49        --  Cartella
stratus_hls           10 settembre 2022, 16:49        --  Cartella
third-party           10 settembre 2022, 16:49        --  Cartella
vivado_hls            10 settembre 2022, 16:49        --  Cartella
```

Figure 4.3: accelerators folder

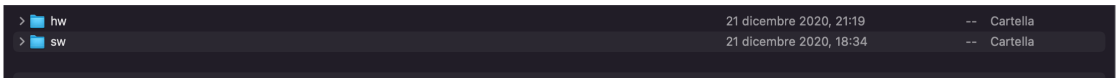
For our purpose we need to go inside the *catapult_hls* folder. Here, as we can see in Fig. 4.4, we have a folder for each accelerator that we have designed following this HLS flow.



common	10 settembre 2022, 16:49	--	Cartella
depthwise_cxx_catapult	18 settembre 2022, 17:23	--	Cartella
softmax_cxx_catapult	oggi, 21:56	--	Cartella

Figure 4.4: *catapult_hls* folder

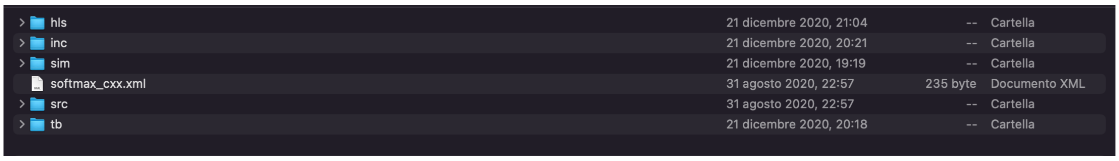
It is important to notice that the name of these directories is standardized in ESP and it is *$ACCELERATOR_cxx_catapult* where the *$ACCELERATOR* variable is the name of our accelerator. In this case we can see that we have the sample *softmax_cxx_catapult* and we have also already created the folder for the new Depthwise accelerator that is simply *depthwise_cxx_catapult*. Instead the *common* folder contains the common files that are shared among all the Catapult accelerators like the ESP header files. In order to see how a complete accelerator is structured, we can go inside the softmax accelerator folder. Here we have two different folders: an hardware folder (*hw*) and a software folder (*sw*) as shown in Fig. 4.5.



hw	21 dicembre 2020, 21:19	--	Cartella
sw	21 dicembre 2020, 18:34	--	Cartella

Figure 4.5: softmax accelerator folder

In this chapter we will analyze only the hardware folder. Finally, moving inside the *hw* folder, we can see the softmax accelerator directory hierarchy, shown in Fig. 4.6.



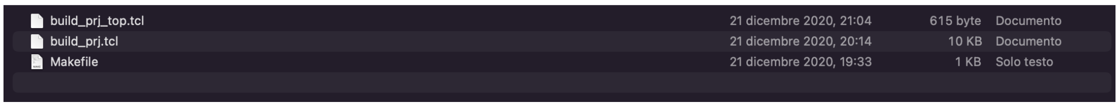
hls	21 dicembre 2020, 21:04	--	Cartella
inc	21 dicembre 2020, 20:21	--	Cartella
sim	21 dicembre 2020, 19:19	--	Cartella
softmax_cxx.xml	31 agosto 2020, 22:57	235 byte	Documento XML
src	31 agosto 2020, 22:57	--	Cartella
tb	21 dicembre 2020, 20:18	--	Cartella

Figure 4.6: *hw* folder

Now we will analyze in details all the files that are contained in these directories, showing how they must be modified in order to realize our accelerator.

4.2.2 *hls* Folder

Inside this folder we have only three files as we can see from Fig. 4.7.

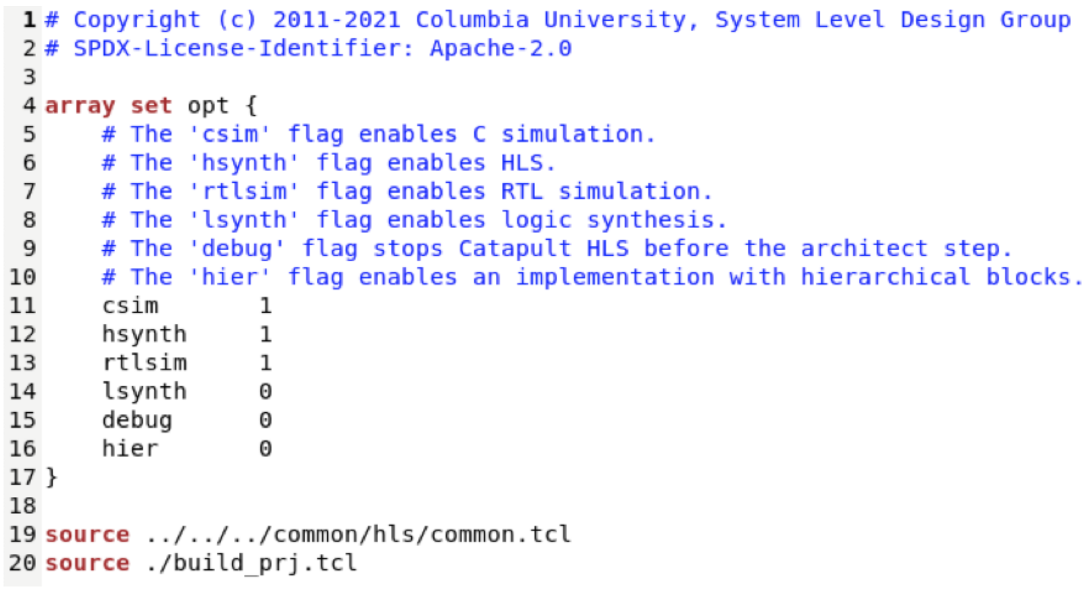


```
build_prj_top.tcl    21 dicembre 2020, 21:04    615 byte    Documento
build_prj.tcl        21 dicembre 2020, 20:14    10 KB       Documento
Makefile             21 dicembre 2020, 19:33    1 KB        Solo testo
```

Figure 4.7: hls folder

build_prj_top.tcl

This file can be used in order to set some useful parameters for the HLS script.



```
# Copyright (c) 2011-2021 Columbia University, System Level Design Group
# SPDX-License-Identifier: Apache-2.0

array set opt {
    # The 'csim' flag enables C simulation.
    # The 'hsynth' flag enables HLS.
    # The 'rtlsim' flag enables RTL simulation.
    # The 'lsynth' flag enables logic synthesis.
    # The 'debug' flag stops Catapult HLS before the architect step.
    # The 'hier' flag enables an implementation with hierarchical blocks.
    csim        1
    hsynth      1
    rtlsim      1
    lsynth      0
    debug       0
    hier        0
}

source ../../../common/hls/common.tcl
source ./build_prj.tcl
```

Figure 4.8: build_prj_top.tcl

In particular as we can see from Fig. 4.8 we can select:

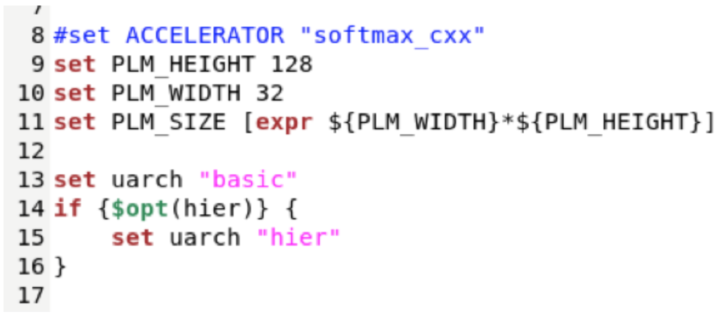
- C simulation with Catapult HLS, asserting the *csim* flag.
- High level synthesis with Catapult HLS, asserting the *hsynth* flag.
- RTL simulation with QuestaSim, asserting the *rtlsim* flag.
- Logic synthesis with Vivado HLS, asserting the *lsynth* flag.

- Debugging of Catapult HLS, asserting the *debug* flag, which means stopping the HLS process ad the architect step.
- Hierarchical implementation of the top level design, asserting the *hier* flag (this will be discussed later in this chapter).

In this file we do not have to do any modification, but we are free to enable the flags value. Finally the last two lines call two other scripts, the *common.tcl* and the *build_prj.tcl*

build_prj.tcl

This is the real script that contains all the Catapult HLS directives used to synthesize the C/C++ code of the accelerator. At the beginning of the file the variables that will set the dimension of the PLMs used by the accelerator are defined. In Fig. 4.9 we can see that in the Softmax example only a single PLM dimension with 128 cells and a bitwidth of 32 bits per cell is needed.



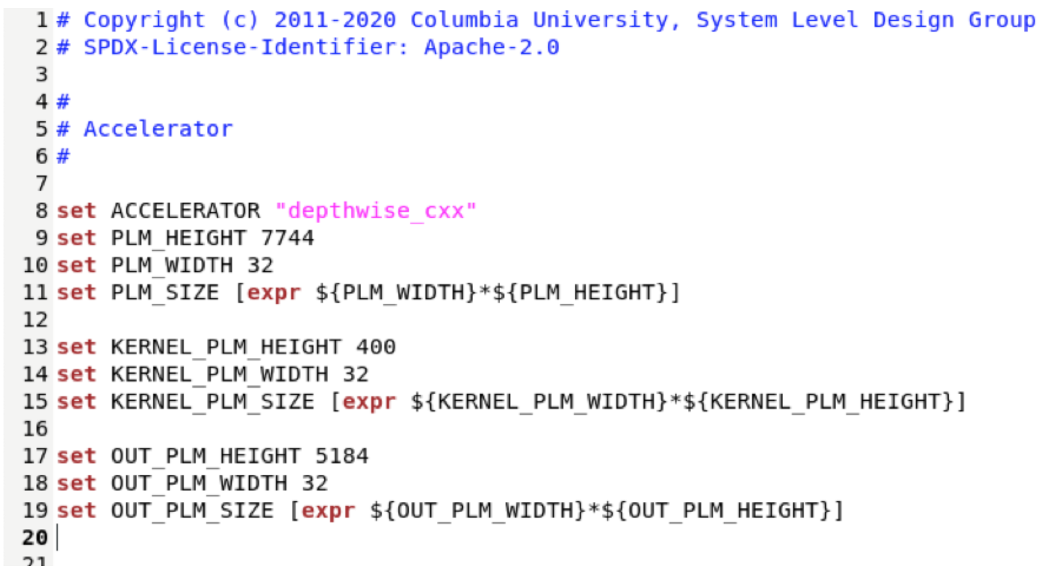
```
#set ACCELERATOR "softmax_cxx"
set PLM_HEIGHT 128
set PLM_WIDTH 32
set PLM_SIZE [expr ${PLM_WIDTH}*${PLM_HEIGHT}]

set uarch "basic"
if {$opt(hier)} {
    set uarch "hier"
}

```

Figure 4.9: softmax build_prj.tcl: PLM dimension

In our case we need three different memory dimensions, one for the PLM that contains the ifmap, one for the PLM that contains the weights and another one for the PLM that contains the ofmap, all using a word width of 32 bits. In order to compute these sizes, we need to use the maximum dimensions that the accelerator memories can reach. In our case we set:

- $IFMAP\_WORDS_{MAX} = (INIT\_WIDTH_{MAX} + PADDING_{MAX})^2 \cdot CHANNELS\_MAX = (16+6)^2 \cdot 16 = 7744$
- $KERNEL\_WORDS_{MAX} = KERNEL\_WIDTH^2_{MAX} \cdot CHANNELS_{MAX} = 5^2 \cdot 16 = 400$
- $OFMAP\_WORDS_{MAX} = \{[(INIT\_WIDTH_{MAX} + PADDING_{MAX} - KERNEL\_WIDTH_{MAX})/STRIDE_{MIN}] + 1\}^2 \cdot CHANNELS_{MAX} = \{[(16+6-5)/1] + 1\}^2 \cdot 16 = 5184$

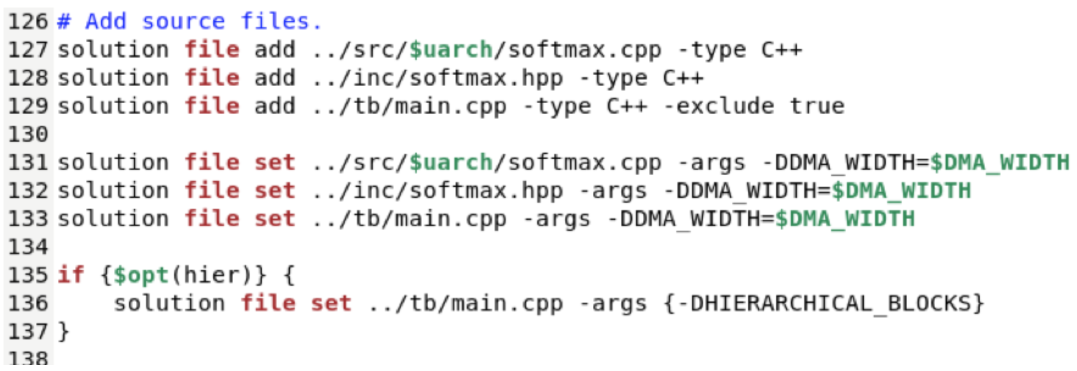
Fig. 4.10 shows the updated memory sized in the build_prj.tcl.



```
# Copyright (c) 2011-2020 Columbia University, System Level Design Group
# SPDX-License-Identifier: Apache-2.0

#
# Accelerator
#

set ACCELERATOR "depthwise_cxx"
set PLM_HEIGHT 7744
set PLM_WIDTH 32
set PLM_SIZE [expr ${PLM_WIDTH}*${PLM_HEIGHT}]

set KERNEL_PLM_HEIGHT 400
set KERNEL_PLM_WIDTH 32
set KERNEL_PLM_SIZE [expr ${KERNEL_PLM_WIDTH}*${KERNEL_PLM_HEIGHT}]

set OUT_PLM_HEIGHT 5184
set OUT_PLM_WIDTH 32
set OUT_PLM_SIZE [expr ${OUT_PLM_WIDTH}*${OUT_PLM_HEIGHT}]

```

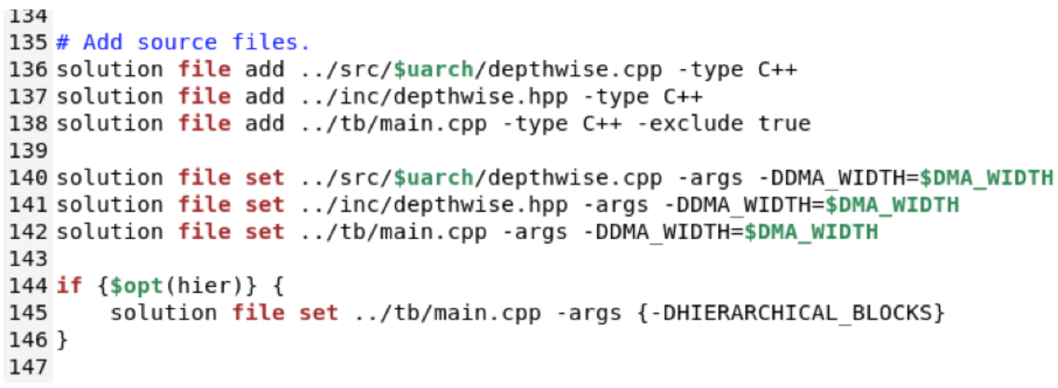
Figure 4.10: depthwise build_prj.tcl: PLM dimension

We need to do other few modifications in this file. Scrolling down in the script we can find the section where all the source files are provided to Catapult, so we have to replace the Softmax input files with our Depthwise input files. In Fig. 4.11 we can see the Softmax example, while in Fig. 4.12 we can see the applied modification.



```
# Add source files.
solution file add ../src/$uarch/softmax.cpp -type C++
solution file add ../inc/softmax.hpp -type C++
solution file add ../tb/main.cpp -type C++ -exclude true

solution file set ../src/$uarch/softmax.cpp -args -DDMA_WIDTH=$DMA_WIDTH
solution file set ../inc/softmax.hpp -args -DDMA_WIDTH=$DMA_WIDTH
solution file set ../tb/main.cpp -args -DDMA_WIDTH=$DMA_WIDTH

if {$opt(hier)} {
    solution file set ../tb/main.cpp -args {-DHIERARCHICAL_BLOCKS}
}
```

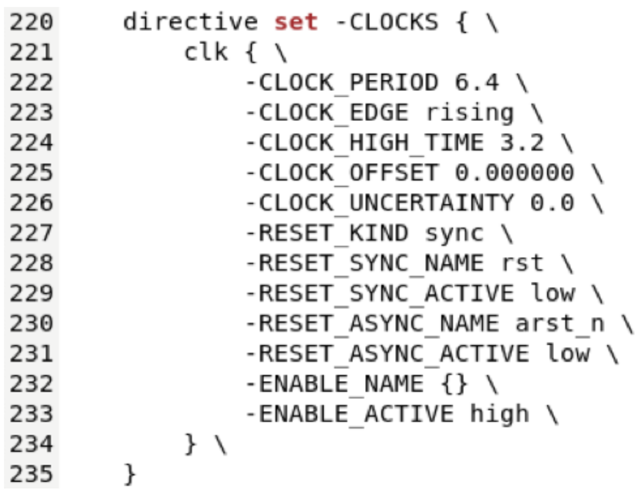
Figure 4.11: softmax build_prj.tcl (input files)



```

# Add source files.
solution file add ../src/$uarch/depthwise.cpp -type C++
solution file add ../inc/depthwise.hpp -type C++
solution file add ../tb/main.cpp -type C++ -exclude true

solution file set ../src/$uarch/depthwise.cpp -args -DDMA_WIDTH=$DMA_WIDTH
solution file set ../inc/depthwise.hpp -args -DDMA_WIDTH=$DMA_WIDTH
solution file set ../tb/main.cpp -args -DDMA_WIDTH=$DMA_WIDTH

if {$opt(hier)} {
    solution file set ../tb/main.cpp -args {-DHIERARCHICAL_BLOCKS}
}

```

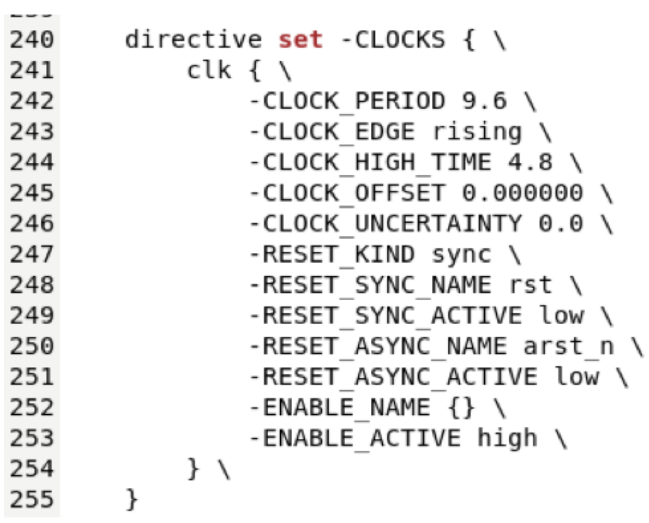
Figure 4.12: depthwise build_prj.tcl: input files

It is possible that during the synthesis some timing errors occur, this is due to the fact that Catapult can not find a feasible solution that satisfy the chosen clock period. A possible solution to overcome this issue is to increase the clock period in the tcl script. For example, we can see in Fig. 4.13 that for the Softmax accelerator the clock period was set to 6.4 *ns*. This period is unfeasible for our Depthwise accelerator so, as we can see in Fig. 4.14 we have increased this value to 9.6 *ns* in order to let the HLS tool find a schedulable solution.



```
    directive set -CLOCKS { \
        clk { \
            -CLOCK_PERIOD 6.4 \
            -CLOCK_EDGE rising \
            -CLOCK_HIGH_TIME 3.2 \
            -CLOCK_OFFSET 0.000000 \
            -CLOCK_UNCERTAINTY 0.0 \
            -RESET_KIND sync \
            -RESET_SYNC_NAME rst \
            -RESET_SYNC_ACTIVE low \
            -RESET_ASYNC_NAME arst_n \
            -RESET_ASYNC_ACTIVE low \
            -ENABLE_NAME {} \
            -ENABLE_ACTIVE high \
        } \
    }
```

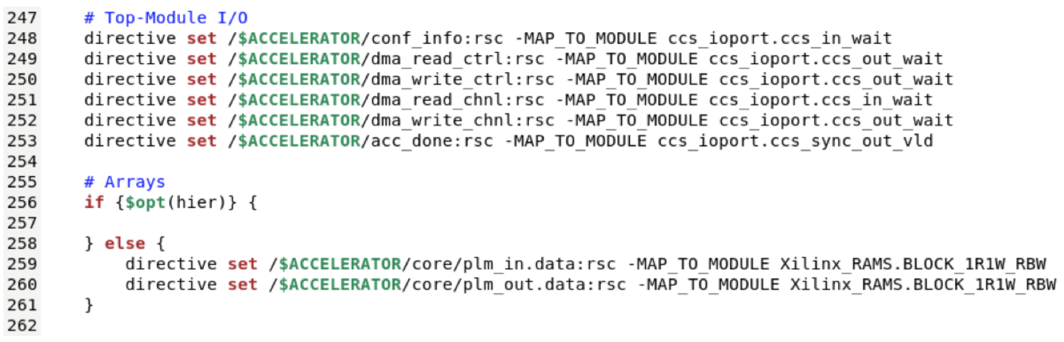
Figure 4.13: softmax build_prj.tcl: clock period



```
directive set -CLOCKS { \
    clk { \
        -CLOCK_PERIOD 9.6 \
        -CLOCK_EDGE rising \
        -CLOCK_HIGH_TIME 4.8 \
        -CLOCK_OFFSET 0.000000 \
        -CLOCK_UNCERTAINTY 0.0 \
        -RESET_KIND sync \
        -RESET_SYNC_NAME rst \
        -RESET_SYNC_ACTIVE low \
        -RESET_ASYNC_NAME arst_n \
        -RESET_ASYNC_ACTIVE low \
        -ENABLE_NAME {} \
        -ENABLE_ACTIVE high \
    } \
}
```

Figure 4.14: depthwise build_prj.tcl: clock period

An interesting section inside this script file is the one regarding the interfaces, under the *#Top-Module I/O* comment.

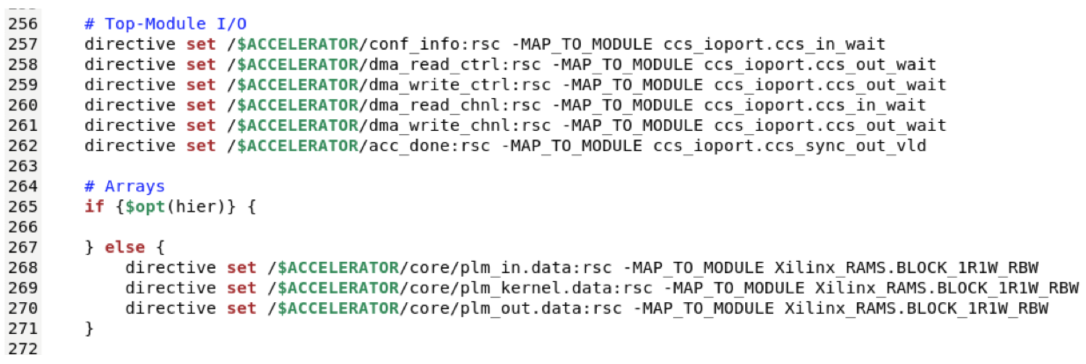


```
# Top-Module I/O
directive set /$ACCELERATOR/conf_info:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set /$ACCELERATOR/dma_read_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /$ACCELERATOR/dma_write_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /$ACCELERATOR/dma_read_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set /$ACCELERATOR/dma_write_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /$ACCELERATOR/acc_done:rsc -MAP_TO_MODULE ccs_ioport.ccs_sync_out_vld

# Arrays
if {$opt(hier)} {

} else {
    directive set /$ACCELERATOR/core/plm_in.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
    directive set /$ACCELERATOR/core/plm_out.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
}

```

Figure 4.15: softmax build_prj.tcl: interfaces and PLMs

In fact, in Fig. 4.15 we can see the definition of the channels and the interfaces of the accelerator, already mentioned in the previous section in Fig. 4.1. In particular, we can notice how they can be modeled in Catapult HLS using the *ccs_ioport* in the *in_wait* and *out_wait* configurations for the input and output channels, respectively, instead for the done signal the *sync_out_vld* configuration is used in order to implement the handshake mechanism when asserting it.
The last section that we have to modify is the *Arrays* section, below the interfaces. Here the required PLMs are physically defined. In Fig. 4.15 we can see that the Softmax accelerator needs only two PLMs, one for the inputs and one for the outputs and both are modeled with a one read/one write RAM block. For the

Depthwise accelerator we need three PLMs, one for the ifmap, one for the kernel and one for the ofmap, as shown in Fig. 4.16.



```
# Top-Module I/O
directive set /$ACCELERATOR/conf_info:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set /$ACCELERATOR/dma_read_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /$ACCELERATOR/dma_write_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /$ACCELERATOR/dma_read_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set /$ACCELERATOR/dma_write_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set /$ACCELERATOR/acc_done:rsc -MAP_TO_MODULE ccs_ioport.ccs_sync_out_vld

# Arrays
if {$opt(hier)} {

} else {
    directive set /$ACCELERATOR/core/plm_in.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
    directive set /$ACCELERATOR/core/plm_kernel.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
    directive set /$ACCELERATOR/core/plm_out.data:rsc -MAP_TO_MODULE Xilinx_RAMS.BLOCK_1R1W_RBW
}

```

Figure 4.16: depthwise build_prj.tcl: interfaces and PLMs

After this modification, the final structure of our Depthwise accelerator will be the one showed in Fig. 4.17.

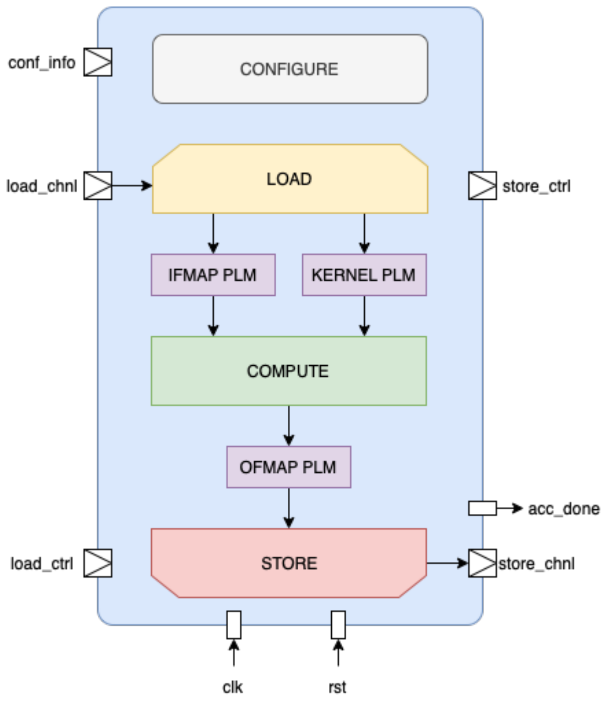


Figure 4.17: depthwise accelerator structure

Makefile

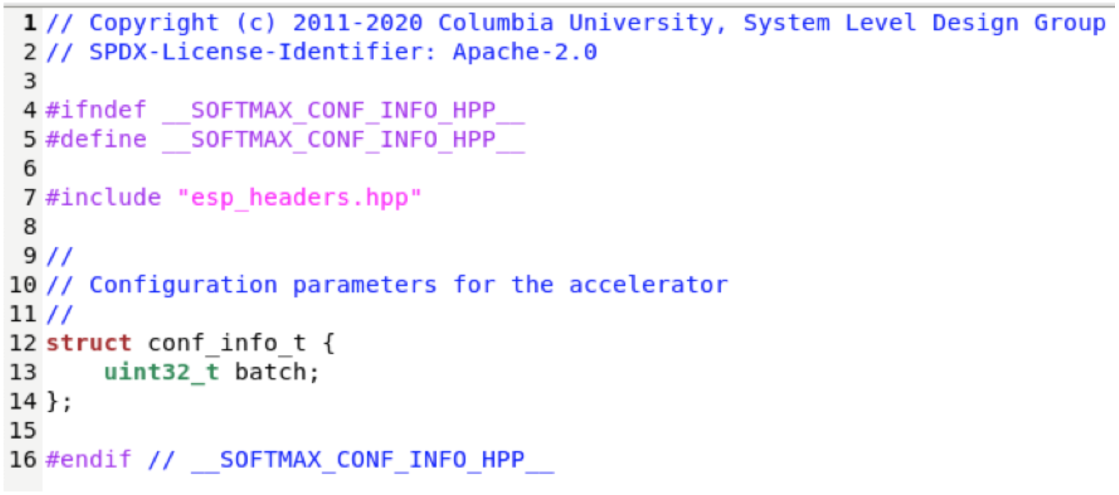
This is a simple file that is used to get the useful source files and parameters for the synthesis process directly from the common directory and we do not need to modify it.

4.2.3 *inc* folder

This folder contains four header files required by the accelerator. In particular these header files declare some useful datatypes that are used in the C++ code of the accelerator.

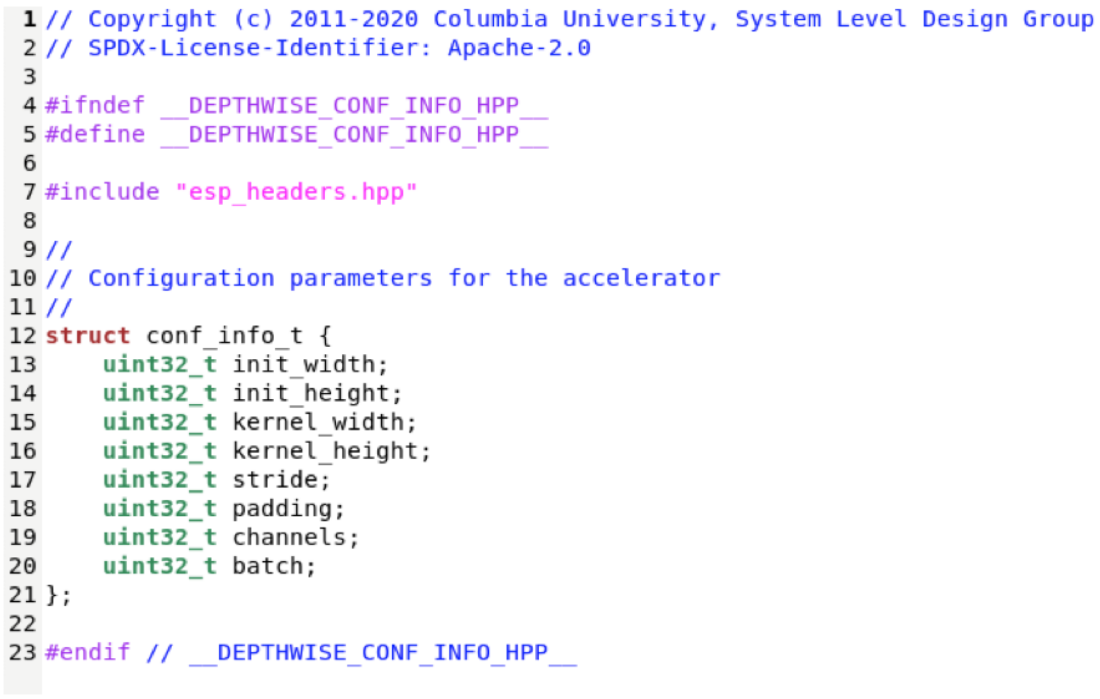
conf_info.hpp

In this header file a C struct called *conf_info_t* is declared. It contains the run-time parameters required by the accelerator. The only modification that we have to do is simply putting inside this structure the parameters that our accelerator needs. In Fig. 4.18, for example, we can see that the Softmax accelerator needs only a *batch* parameter while in 4.19 inside the *conf_info_t* structure there are the Depthwise parameters.



```
// Copyright (c) 2011-2020 Columbia University, System Level Design Group
// SPDX-License-Identifier: Apache-2.0

#ifndef __SOFTMAX_CONF_INFO_HPP__
#define __SOFTMAX_CONF_INFO_HPP__

#include "esp_headers.hpp"

//
// Configuration parameters for the accelerator
//
struct conf_info_t {
    uint32_t batch;
};

#endif // __SOFTMAX_CONF_INFO_HPP__
```

Figure 4.18: softmax conf_info.hpp



```
// Copyright (c) 2011-2020 Columbia University, System Level Design Group
// SPDX-License-Identifier: Apache-2.0

#ifndef __DEPTHWISE_CONF_INFO_HPP__
#define __DEPTHWISE_CONF_INFO_HPP__

#include "esp_headers.hpp"

//
// Configuration parameters for the accelerator
//
struct conf_info_t {
    uint32_t init_width;
    uint32_t init_height;
    uint32_t kernel_width;
    uint32_t kernel_height;
    uint32_t stride;
    uint32_t padding;
    uint32_t channels;
    uint32_t batch;
};

#endif // __DEPTHWISE_CONF_INFO_HPP__
```

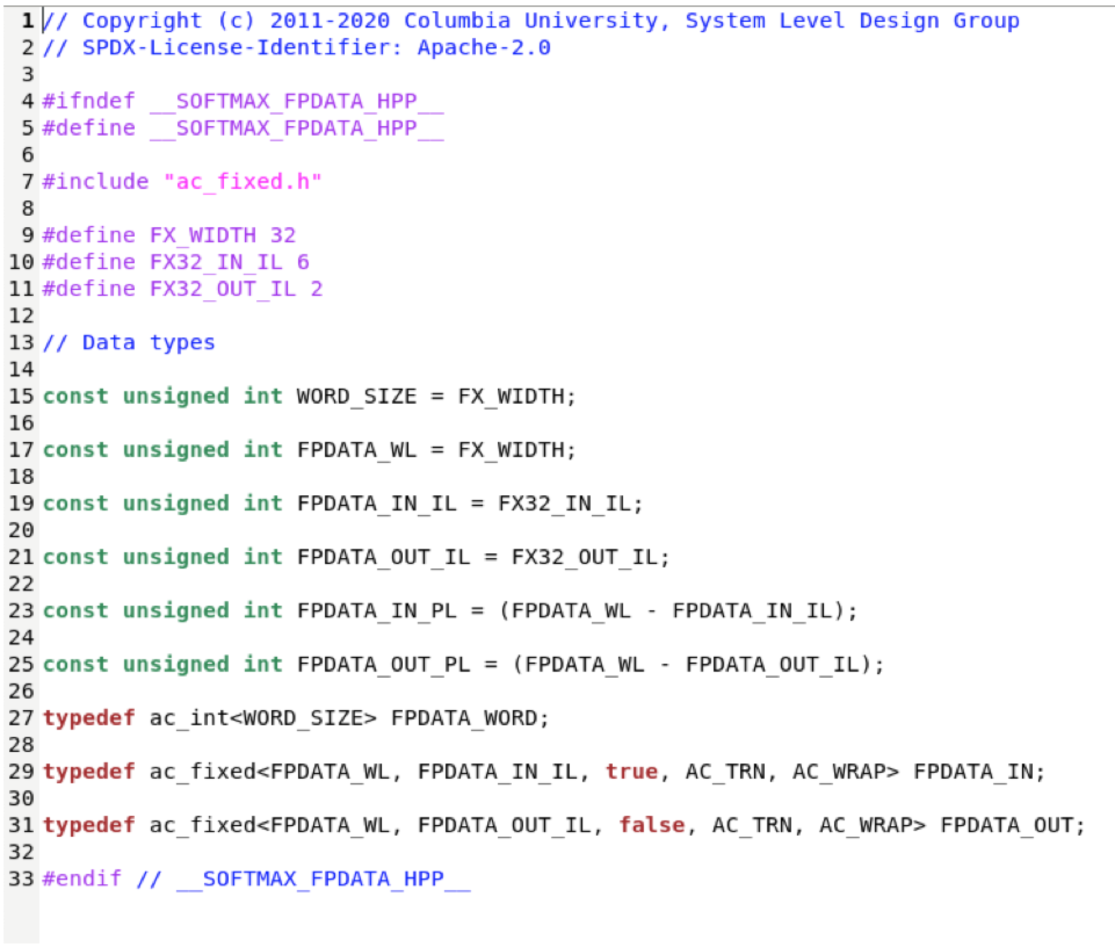
Figure 4.19: depthwise conf_info.hpp

debug_info.hpp

This header file contains only a C struct called *debug_info_t* but in this case we do not need to do any modification.

fpdata.hpp

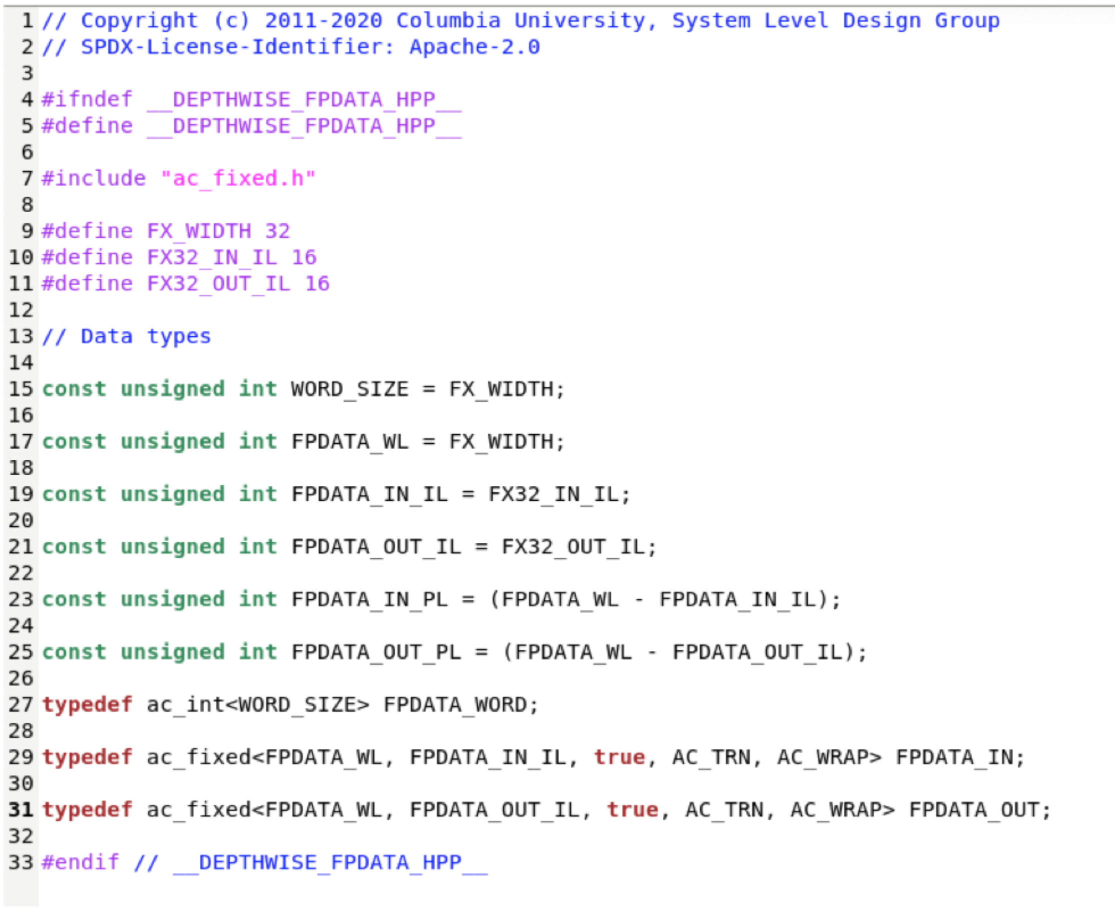
In this header file some useful datatypes that are used to represent fixed point numbers inside the accelerator are declared. In order to declare these variables the *algoritmic_C* datatypes from the *ac_fixed.h* library are used. The *algoritmic_C* datatype is a specific set of Catapult HLS datatypes used to represent numbers and interfaces [18]. In our case, as we can see from Fig. 4.20 for the Softmax accelerator, we only use the *ac_fixed* and *ac_int* datatypes to define our fixed point data.



```
// Copyright (c) 2011-2020 Columbia University, System Level Design Group
// SPDX-License-Identifier: Apache-2.0

#ifndef __SOFTMAX_FPDATA_HPP__
#define __SOFTMAX_FPDATA_HPP__

#include "ac_fixed.h"

#define FX_WIDTH 32
#define FX32_IN_IL 6
#define FX32_OUT_IL 2

// Data types

const unsigned int WORD_SIZE = FX_WIDTH;

const unsigned int FPDATA_WL = FX_WIDTH;

const unsigned int FPDATA_IN_IL = FX32_IN_IL;

const unsigned int FPDATA_OUT_IL = FX32_OUT_IL;

const unsigned int FPDATA_IN_PL = (FPDATA_WL - FPDATA_IN_IL);

const unsigned int FPDATA_OUT_PL = (FPDATA_WL - FPDATA_OUT_IL);

typedef ac_int<WORD_SIZE> FPDATA_WORD;

typedef ac_fixed<FPDATA_WL, FPDATA_IN_IL, true, AC_TRN, AC_WRAP> FPDATA_IN;

typedef ac_fixed<FPDATA_WL, FPDATA_OUT_IL, false, AC_TRN, AC_WRAP> FPDATA_OUT;

#endif // __SOFTMAX_FPDATA_HPP__
```

Figure 4.20: softmax fpdata.hpp

In particular, in the example, we notice that the word length is set to 32 bits with the *FX_WIDTH* define. Then with *FX_IN_IL* and *FX_OUT_IL* the length for the integer part of the input and output numbers is set,respectively. In this way it is possible to have two different representations for inputs and outputs. It is important to notice the declaration of the *FPDATA_IN* and *FPDATA_OUT* datatypes. Here it is used the *ac_fixed* datatype where we can specify the total bitwidth, the length of the integer part, if the number must be considered with (true) or without sign (false), which are the rounding and overflow mechanisms. For our Depthwise accelerator, as we can see in Fig 4.21 we have modified the integer length setting both the inputs and outputs to 16 and considering both signed numbers.

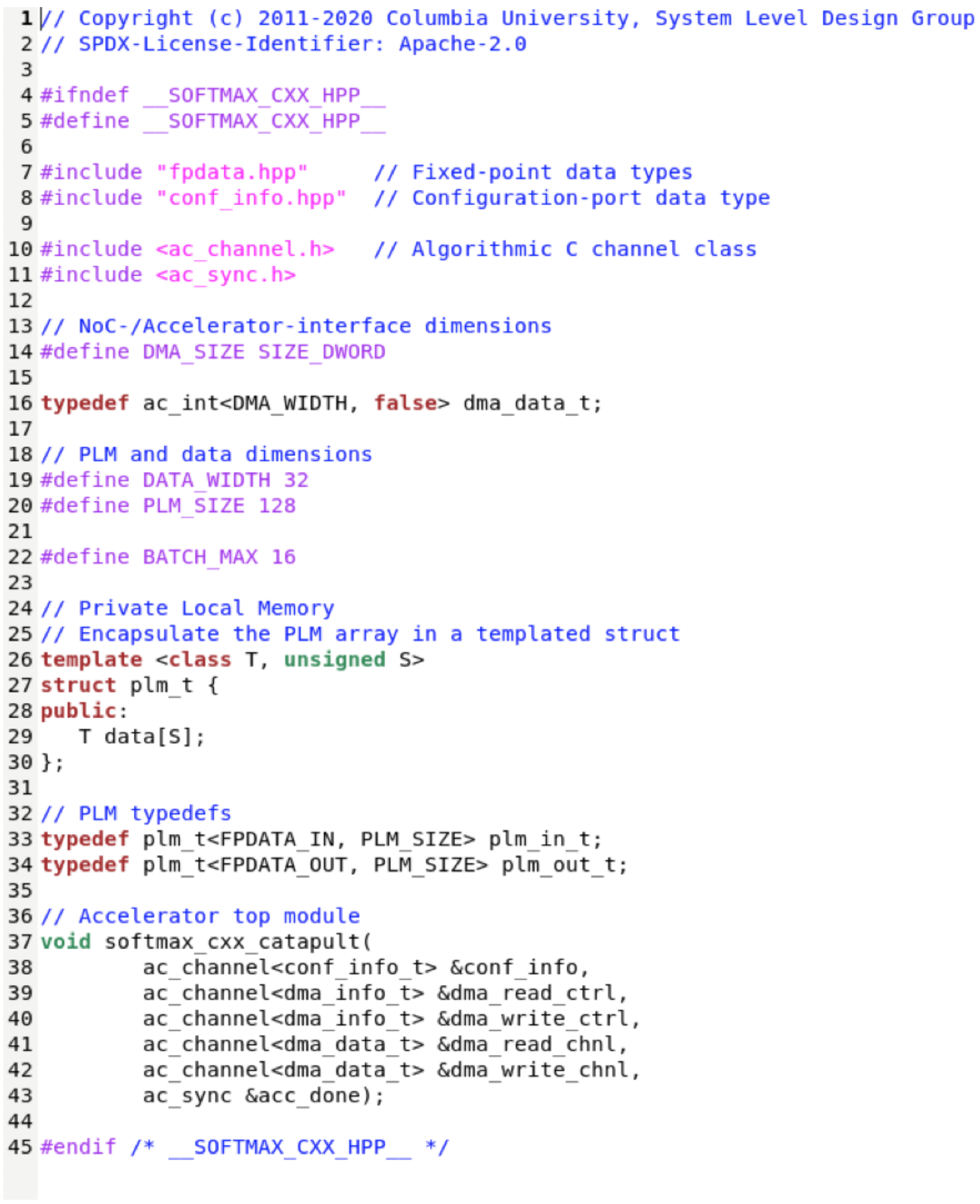


```
// Copyright (c) 2011-2020 Columbia University, System Level Design Group
// SPDX-License-Identifier: Apache-2.0

#ifndef __DEPTHWISE_FPDATA_HPP__
#define __DEPTHWISE_FPDATA_HPP__

#include "ac_fixed.h"

#define FX_WIDTH 32
#define FX32_IN_IL 16
#define FX32_OUT_IL 16

// Data types

const unsigned int WORD_SIZE = FX_WIDTH;

const unsigned int FPDATA_WL = FX_WIDTH;

const unsigned int FPDATA_IN_IL = FX32_IN_IL;

const unsigned int FPDATA_OUT_IL = FX32_OUT_IL;

const unsigned int FPDATA_IN_PL = (FPDATA_WL - FPDATA_IN_IL);

const unsigned int FPDATA_OUT_PL = (FPDATA_WL - FPDATA_OUT_IL);

typedef ac_int<WORD_SIZE> FPDATA_WORD;

typedef ac_fixed<FPDATA_WL, FPDATA_IN_IL, true, AC_TRN, AC_WRAP> FPDATA_IN;

typedef ac_fixed<FPDATA_WL, FPDATA_OUT_IL, true, AC_TRN, AC_WRAP> FPDATA_OUT;

#endif // __DEPTHWISE_FPDATA_HPP__
```

Figure 4.21: depthwise fpdata.hpp

softmax.hpp/depthwise.hpp

This header file is used to declare the DMA datatypes, the PLMs datatypes and the interfaces of the accelerator's top module using the *algorithmic_C* datatypes. We can see the entire file for the Softmax example in Fig. 4.22

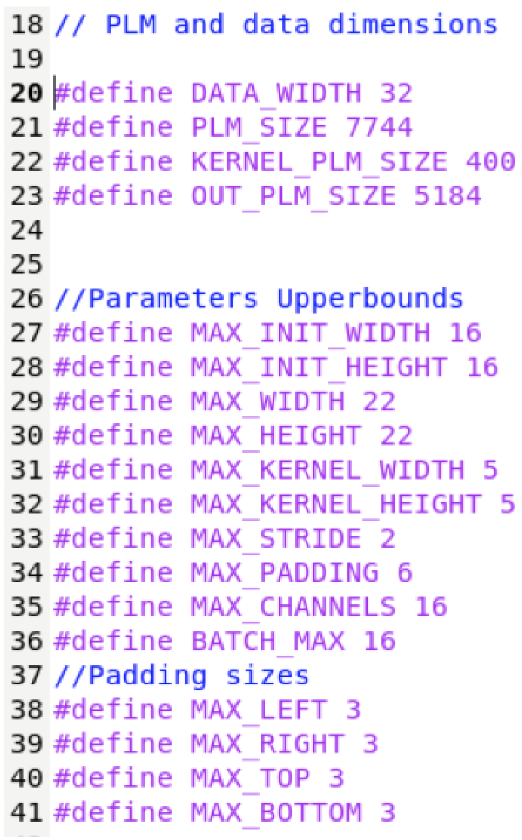


```
// Copyright (c) 2011-2020 Columbia University, System Level Design Group
// SPDX-License-Identifier: Apache-2.0

#ifndef __SOFTMAX_CXX_HPP__
#define __SOFTMAX_CXX_HPP__

#include "fpdata.hpp"      // Fixed-point data types
#include "conf_info.hpp"   // Configuration-port data type

#include <ac_channel.h>    // Algorithmic C channel class
#include <ac_sync.h>

// NoC-/Accelerator-interface dimensions
#define DMA_SIZE SIZE_DWORD

typedef ac_int<DMA_WIDTH, false> dma_data_t;

// PLM and data dimensions
#define DATA_WIDTH 32
#define PLM_SIZE 128

#define BATCH_MAX 16

// Private Local Memory
// Encapsulate the PLM array in a templated struct
template <class T, unsigned S>
struct plm_t {
public:
   T data[S];
};

// PLM typedefs
typedef plm_t<FPDATA_IN, PLM_SIZE> plm_in_t;
typedef plm_t<FPDATA_OUT, PLM_SIZE> plm_out_t;

// Accelerator top module
void softmax_cxx_catapult(
        ac_channel<conf_info_t> &conf_info,
        ac_channel<dma_info_t> &dma_read_ctrl,
        ac_channel<dma_info_t> &dma_write_ctrl,
        ac_channel<dma_data_t> &dma_read_chnl,
        ac_channel<dma_data_t> &dma_write_chnl,
        ac_sync &acc_done);

#endif /* __SOFTMAX_CXX_HPP__ */
```

Figure 4.22: softmax.hpp

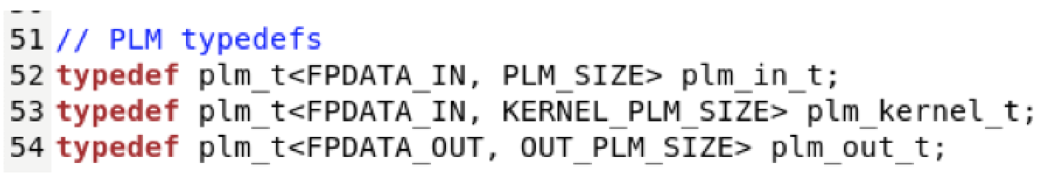
First of all we can see the declaration of the *dma_data_t* datatype using the *ac_int*. Within this DMA datatype, the *DMA_WIDTH* parameter will be set from the command line by the user before executing the *.tcl* file. Then with some *DEFINE* directives the maximum values of the accelerator parameters and the size of the PLMs are defined. After that, we have the declaration of the generic

PLM C struct using the template class approach in order to let the user choose the datatypes for the data inside each memory. Then we can see the actual PLMs declarations. For the Softmax accelerator only two memories with the same size are declared, one for the inputs *plm_in_t* and one for the outputs *plm_out_t*, using the *FPDATA_IN* and *FPDATA_OUT* declared in the *fpdata.hpp* header file. Finally, in the last section the real function that implements the accelerator top module is declared. This is the function that we have to call in software when we want to run our accelerator in baremetal. The arguments of this function are all the interfaces declared using the *ac_channel* and *ac_sync* datatypes. In particular, the *ac_channels* are used for the data that will be transferred with DMA, while the *ac_sync* is only used for the done signal. The only modifications that we have to do in this file are related to the several *DEFINE* directives and the declaration of the PLMs. In Fig. 4.23 we can see our Depthwise maximum values for the convolution parameters, while in Fig. 4.24 we have the declaration of the new additional kernel PLM.



```
// PLM and data dimensions

#define DATA_WIDTH 32
#define PLM_SIZE 7744
#define KERNEL_PLM_SIZE 400
#define OUT_PLM_SIZE 5184


//Parameters Upperbounds
#define MAX_INIT_WIDTH 16
#define MAX_INIT_HEIGHT 16
#define MAX_WIDTH 22
#define MAX_HEIGHT 22
#define MAX_KERNEL_WIDTH 5
#define MAX_KERNEL_HEIGHT 5
#define MAX_STRIDE 2
#define MAX_PADDING 6
#define MAX_CHANNELS 16
#define BATCH_MAX 16
//Padding sizes
#define MAX_LEFT 3
#define MAX_RIGHT 3
#define MAX_TOP 3
#define MAX_BOTTOM 3
```

Figure 4.23: depthwise.hpp: Maximum convolution parameters



```
// PLM typedefs
typedef plm_t<FPDATA_IN, PLM_SIZE> plm_in_t;
typedef plm_t<FPDATA_IN, KERNEL_PLM_SIZE> plm_kernel_t;
typedef plm_t<FPDATA_OUT, OUT_PLM_SIZE> plm_out_t;
```

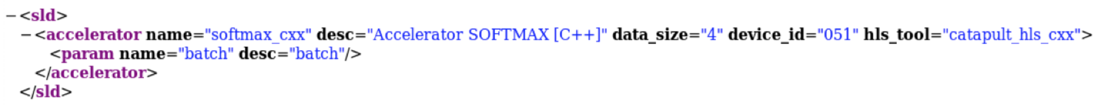
Figure 4.24: depthwise.hpp: PLMs declaration

4.2.4 *sim* folder

In this folder there is only a Makefile that define some variables with useful paths for the script execution. We do not need to modify it.

4.2.5 softmax_cxx.xml/depthwise_cxx.xml

In this file some variables that are useful for the ESP SoC integration of our accelerator are defined.

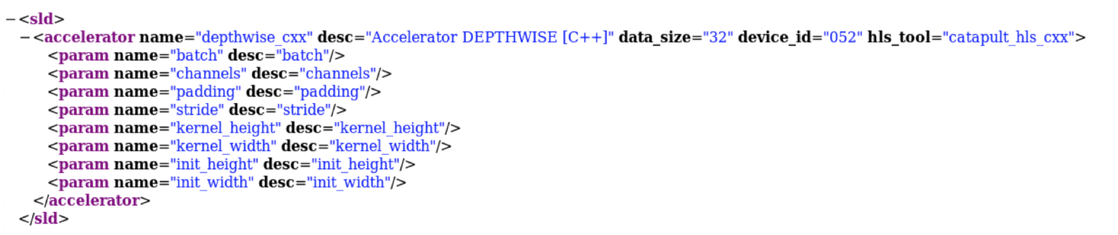


```
- <sld>
  - <accelerator name="softmax_cxx" desc="Accelerator SOFTMAX [C++]" data_size="4" device_id="051" hls_tool="catapult_hls_cxx">
      <param name="batch" desc="batch"/>
    </accelerator>
  </sld>
```

Figure 4.25: softmax_cxx.xml

In Fig. 4.25 we can see that, for the Softmax accelerator are defined:

- The name of the accelerator that will be used in ESP
- A brief description of the accelerator
- The data size
- A unique ID that is used in ESP to identify the accelerator
- The HLS tool used for the accelerator description
- The name of the parameters that are passed to the accelerator



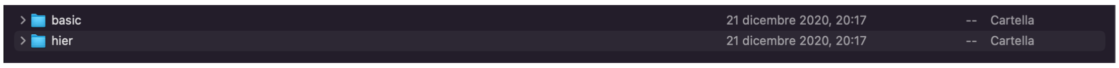
```
– <sld>
  – <accelerator name="depthwise_cxx" desc="Accelerator DEPTHWISE [C++]" data_size="32" device_id="052" hls_tool="catapult_hls_cxx">
      <param name="batch" desc="batch"/>
      <param name="channels" desc="channels"/>
      <param name="padding" desc="padding"/>
      <param name="stride" desc="stride"/>
      <param name="kernel_height" desc="kernel_height"/>
      <param name="kernel_width" desc="kernel_width"/>
      <param name="init_height" desc="init_height"/>
      <param name="init_width" desc="init_width"/>
    </accelerator>
  </sld>
```

Figure 4.26: depthwise_cxx.xml

These fields are common to all the accelerators. In fact in Fig. 4.26 we can see the modifications needed to adapt this file to our Depthwise accelerator.

4.2.6 *src* folder

In Fig. 4.27 we can see that inside this folder there are other two folders, one that is called *basic* and the other one that is called *hier*.



```
> basic    21 dicembre 2020, 20:17    --   Cartella
> hier     21 dicembre 2020, 20:17    --   Cartella
```

Figure 4.27: src folder

This is because, with Catapult HLS, we can design an accelerator following two different architectures:

- **Basic Block Architecture:** the accelerator phases are executed sequentially. After the initial "Configure" phase, the "Load", "Compute" and "Store" phases are all executed in sequence. It means that after a "Load" we will have the "Compute" and then the "Store". After the "Store" phase, if the accelerator works on multiple batches we will have another "Load" and so on. This behavior is summarized in Fig. 4.28.

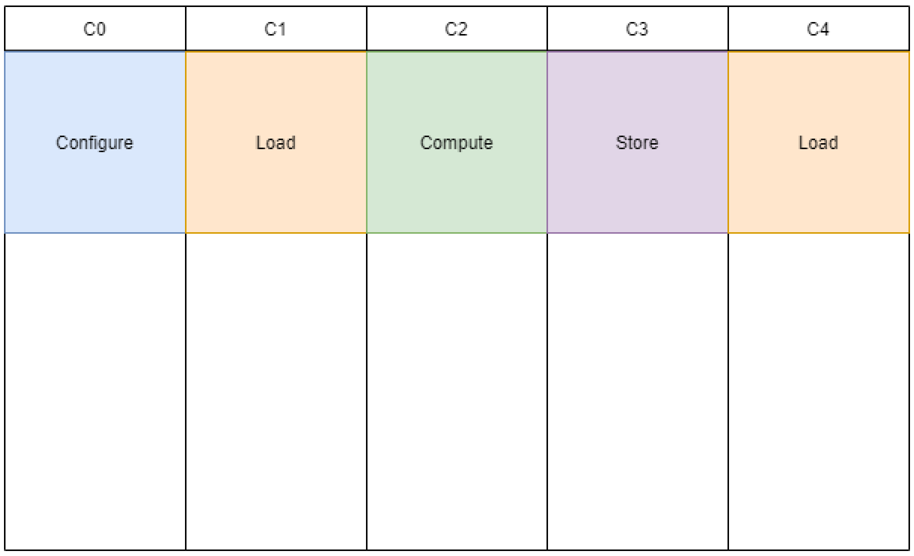


Figure 4.28: Basic Block Architecture

- **Hierarchical Block Architecture:** the accelerator phases are executed in a pipelined way. After the initial "Configure" phase, the "Load", "Compute" and "Store" phases are all executed as in a pipeline. It means that after a "Load" we will have the "Compute" and in the meanwhile also the "Load" of the next batch, then we will have the "Store" and the "Compute" of the second batch, but also the "Load" of the third batch and so on. We can see how this hierarchical architecture works in Fig. 4.29. This is a big improvement in terms of speed with respect to the basic architecture. Clearly we need to take into account also an increment in terms of area due to the additional logic to support this more complex behavior. It is important to notice that we can appreciate this speed increment only if our accelerator supports a computation in multiple batches, otherwise there are no differences in terms of speed between the two architectures and we will see only an unwanted area increment.

Inside both *basic* and *hier* folder there is a C++ source code file that implements, for each architecture, the accelerator top module function declared previously in the accelerator header file.

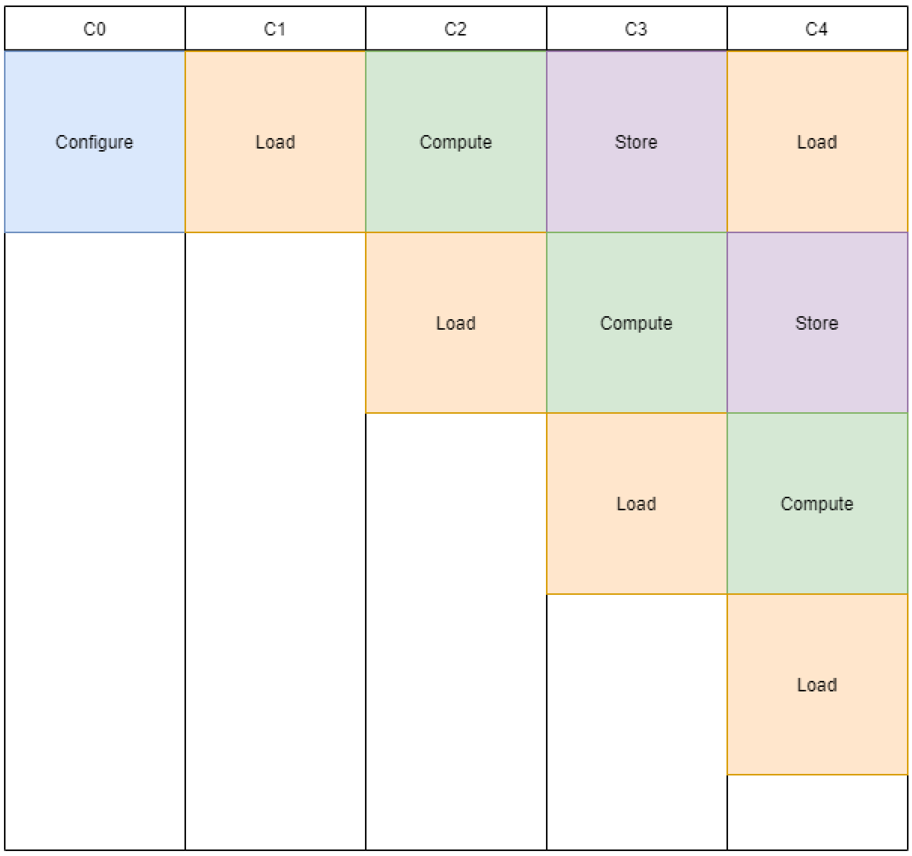


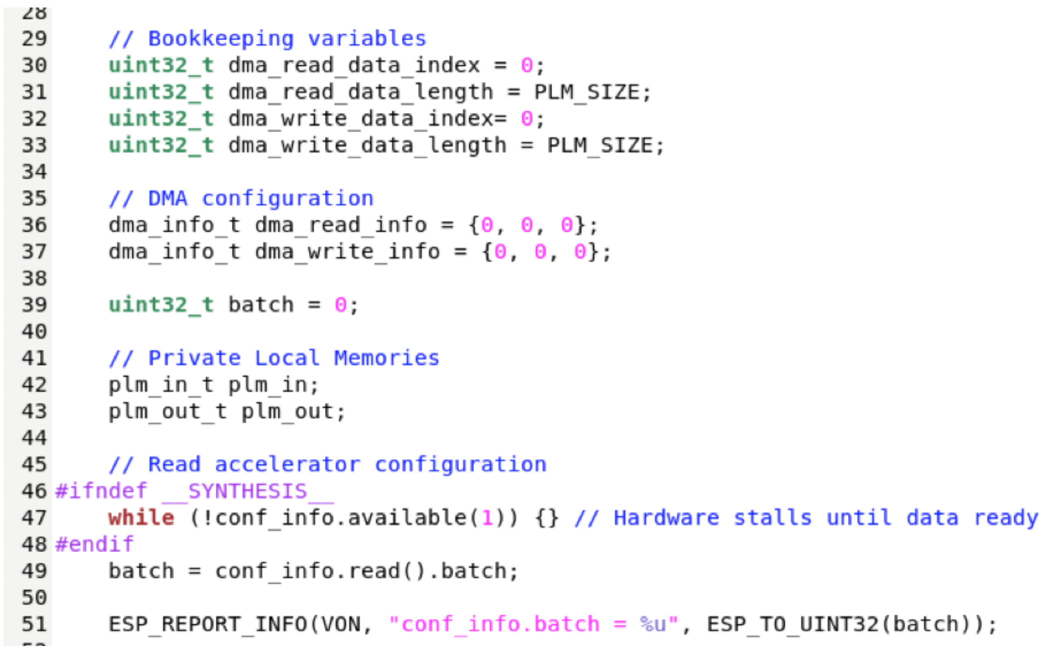
Figure 4.29: Hierarchical Block Architecture

softmax.cpp/depthwise.cpp (basic)

The accelerator top module function starts with the definition of some useful variables used to setup the DMA data transfer.

In particular, as we can see from Fig. 4.30, we have the following variables:

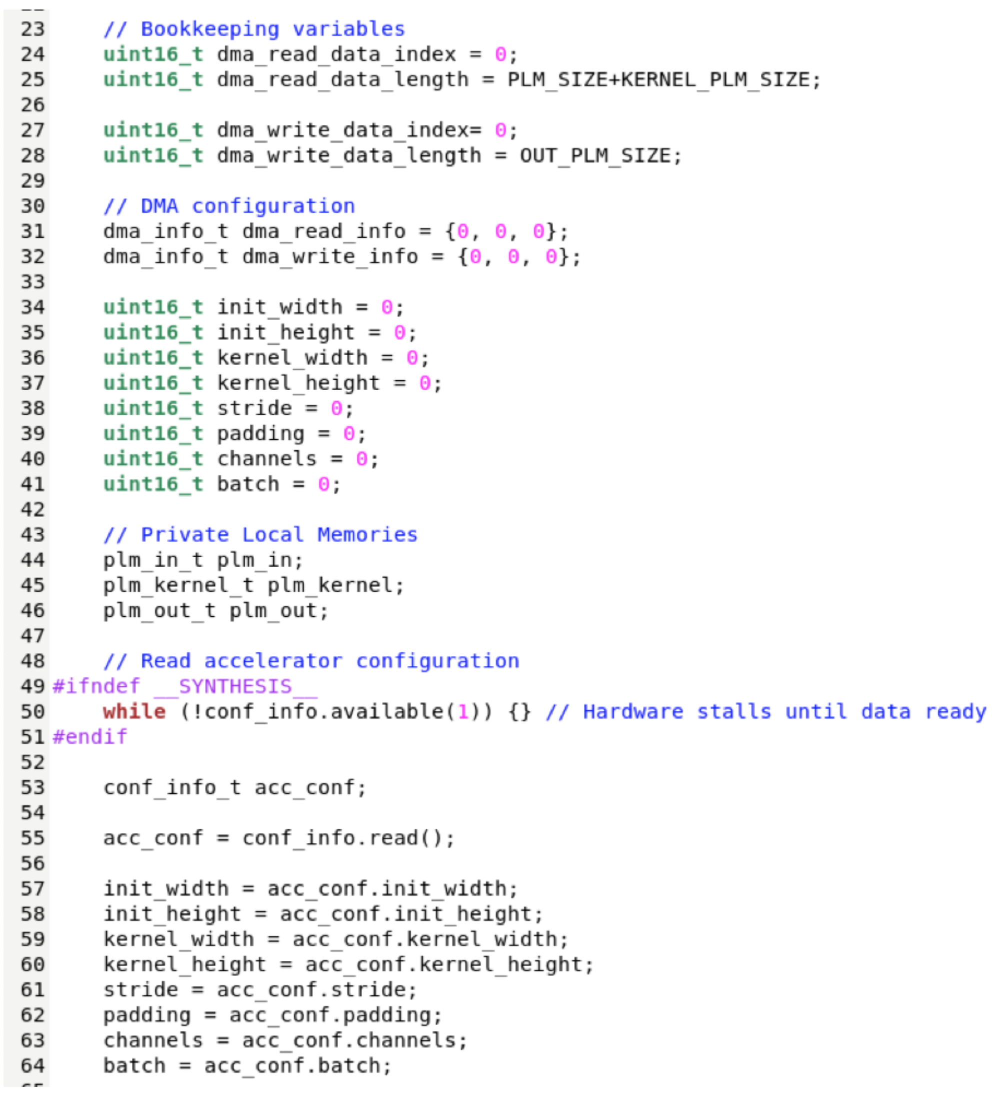
- ***dma_read_data_index*:** used to set the offset from which we want to start reading the external memory using the DMA.
- ***dma_read_data_length*:** used to set the number of readings that we want to perform.
- ***dma_write_data_index*:** used to set the offset from which we want to start writing in the external memory using the DMA.
- ***dma_write_data_length*:** used to set the number of writings that we want to perform.



```
29     // Bookkeeping variables
30     uint32_t dma_read_data_index = 0;
31     uint32_t dma_read_data_length = PLM_SIZE;
32     uint32_t dma_write_data_index= 0;
33     uint32_t dma_write_data_length = PLM_SIZE;
34
35     // DMA configuration
36     dma_info_t dma_read_info = {0, 0, 0};
37     dma_info_t dma_write_info = {0, 0, 0};
38
39     uint32_t batch = 0;
40
41     // Private Local Memories
42     plm_in_t plm_in;
43     plm_out_t plm_out;
44
45     // Read accelerator configuration
46 #ifndef __SYNTHESIS__
47     while (!conf_info.available(1)) {} // Hardware stalls until data ready
48 #endif
49     batch = conf_info.read().batch;
50
51     ESP_REPORT_INFO(VON, "conf_info.batch = %u", ESP_TO_UINT32(batch));
```

Figure 4.30: softmax.cpp: Configure

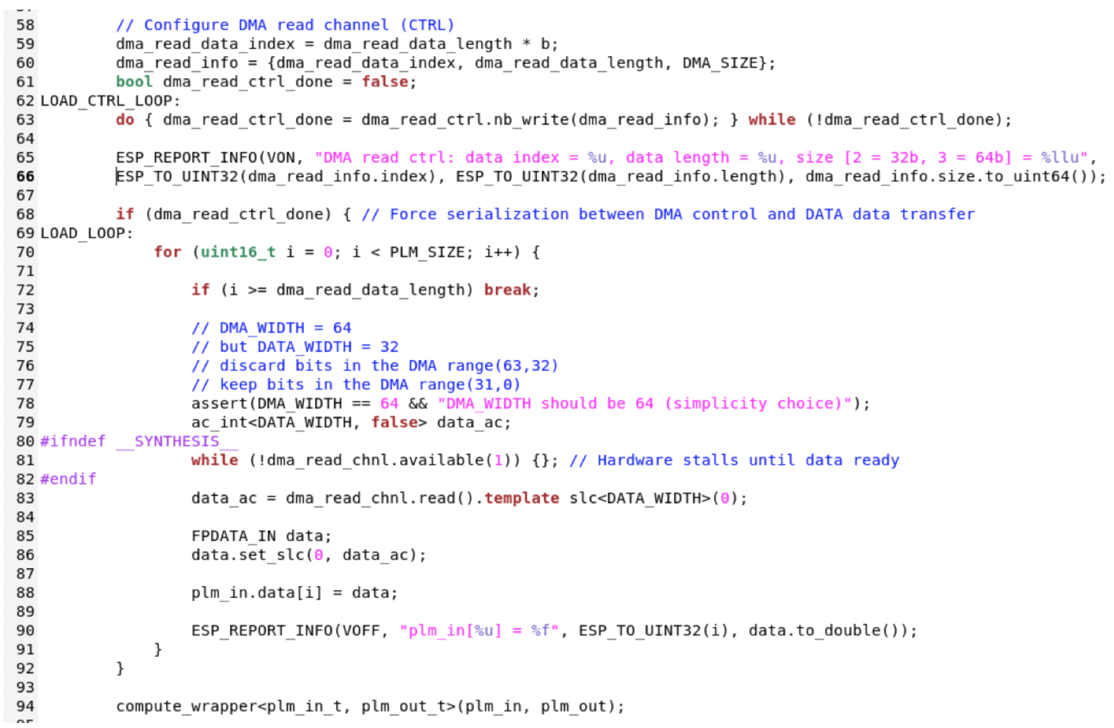
After the PLMs definition we can see the "Configure" phase at line 49 where we read our accelerator parameters value using the *read()* function applied to the *conf_info* structure. For the Softmax example we need only to set the number of batches. Fig. 4.31 shows how we have to modify this section in order to adapt the "Configure" phase to our Depthwise accelerator.



```
    // Bookkeeping variables
    uint16_t dma_read_data_index = 0;
    uint16_t dma_read_data_length = PLM_SIZE+KERNEL_PLM_SIZE;

    uint16_t dma_write_data_index= 0;
    uint16_t dma_write_data_length = OUT_PLM_SIZE;

    // DMA configuration
    dma_info_t dma_read_info = {0, 0, 0};
    dma_info_t dma_write_info = {0, 0, 0};

    uint16_t init_width = 0;
    uint16_t init_height = 0;
    uint16_t kernel_width = 0;
    uint16_t kernel_height = 0;
    uint16_t stride = 0;
    uint16_t padding = 0;
    uint16_t channels = 0;
    uint16_t batch = 0;

    // Private Local Memories
    plm_in_t plm_in;
    plm_kernel_t plm_kernel;
    plm_out_t plm_out;

    // Read accelerator configuration
#ifndef __SYNTHESIS__
    while (!conf_info.available(1)) {} // Hardware stalls until data ready
#endif

    conf_info_t acc_conf;

    acc_conf = conf_info.read();

    init_width = acc_conf.init_width;
    init_height = acc_conf.init_height;
    kernel_width = acc_conf.kernel_width;
    kernel_height = acc_conf.kernel_height;
    stride = acc_conf.stride;
    padding = acc_conf.padding;
    channels = acc_conf.channels;
    batch = acc_conf.batch;
```

Figure 4.31: depthwise.cpp: Configure

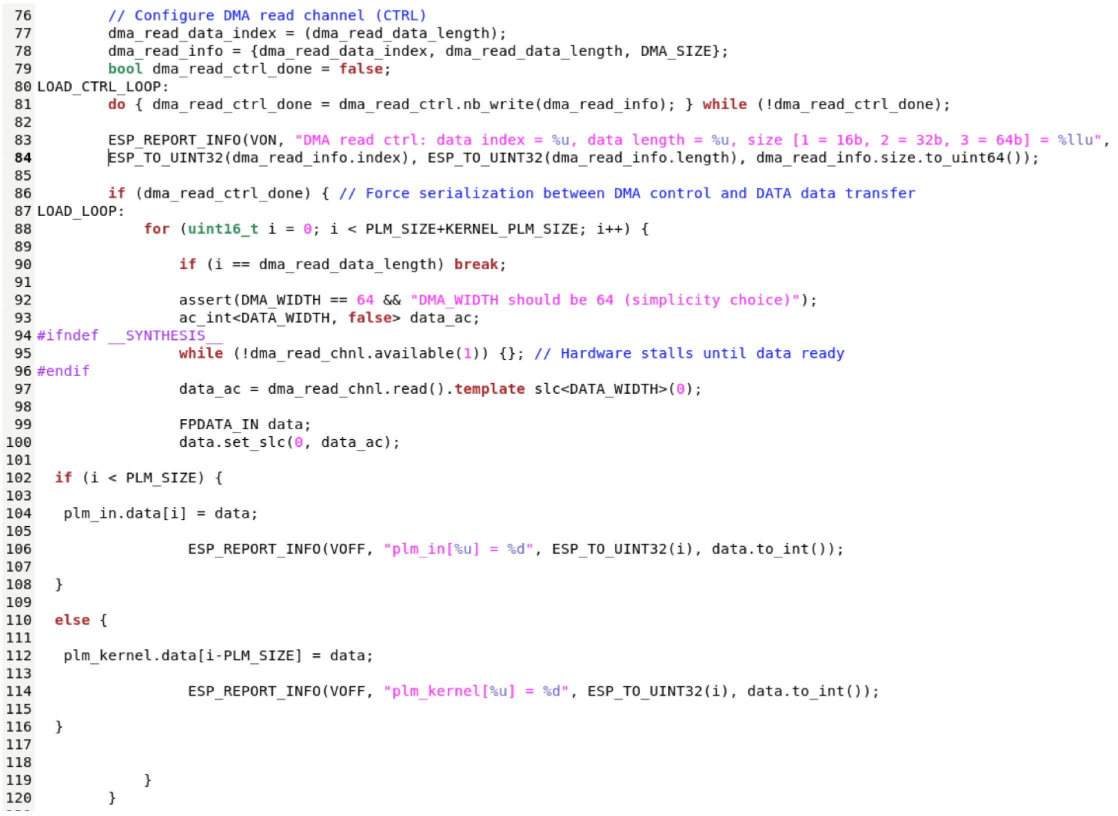
After the "Configure" phase we need to set the DMA variables in order to perform the "Load" phase correctly. So we need a loop that checks that the DMA is correctly configured using a variable called *dma_read_ctrl_done* and then we can proceed with the "Load" phase. In the "Load" phase we simply use the *read()* function on the *dma_read_chnl* in order to read data coming from the DMA channel and store them into our input PLM. We can see this process in Fig. 4.32 for the Softmax accelerator. In this case the DMA width is considered equal to 64 bits.



```
        // Configure DMA read channel (CTRL)
        dma_read_data_index = dma_read_data_length * b;
        dma_read_info = {dma_read_data_index, dma_read_data_length, DMA_SIZE};
        bool dma_read_ctrl_done = false;
LOAD_CTRL_LOOP:
        do { dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info); } while (!dma_read_ctrl_done);

        ESP_REPORT_INFO(VON, "DMA read ctrl: data index = %u, data length = %u, size [2 = 32b, 3 = 64b] = %llu",
        ESP_TO_UINT32(dma_read_info.index), ESP_TO_UINT32(dma_read_info.length), dma_read_info.size.to_uint64());

        if (dma_read_ctrl_done) { // Force serialization between DMA control and DATA data transfer
LOAD_LOOP:
            for (uint16_t i = 0; i < PLM_SIZE; i++) {

                if (i >= dma_read_data_length) break;

                // DMA_WIDTH = 64
                // but DATA_WIDTH = 32
                // discard bits in the DMA range(63,32)
                // keep bits in the DMA range(31,0)
                assert(DMA_WIDTH == 64 && "DMA_WIDTH should be 64 (simplicity choice)");
                ac_int<DATA_WIDTH, false> data_ac;
#ifndef __SYNTHESIS__
                while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data ready
#endif
                data_ac = dma_read_chnl.read().template slc<DATA_WIDTH>(0);

                FPDATA_IN data;
                data.set_slc(0, data_ac);

                plm_in.data[i] = data;

                ESP_REPORT_INFO(VOFF, "plm_in[%u] = %f", ESP_TO_UINT32(i), data.to_double());
            }
        }

        compute_wrapper<plm_in_t, plm_out_t>(plm_in, plm_out);
```

Figure 4.32: softmax.cpp: Load

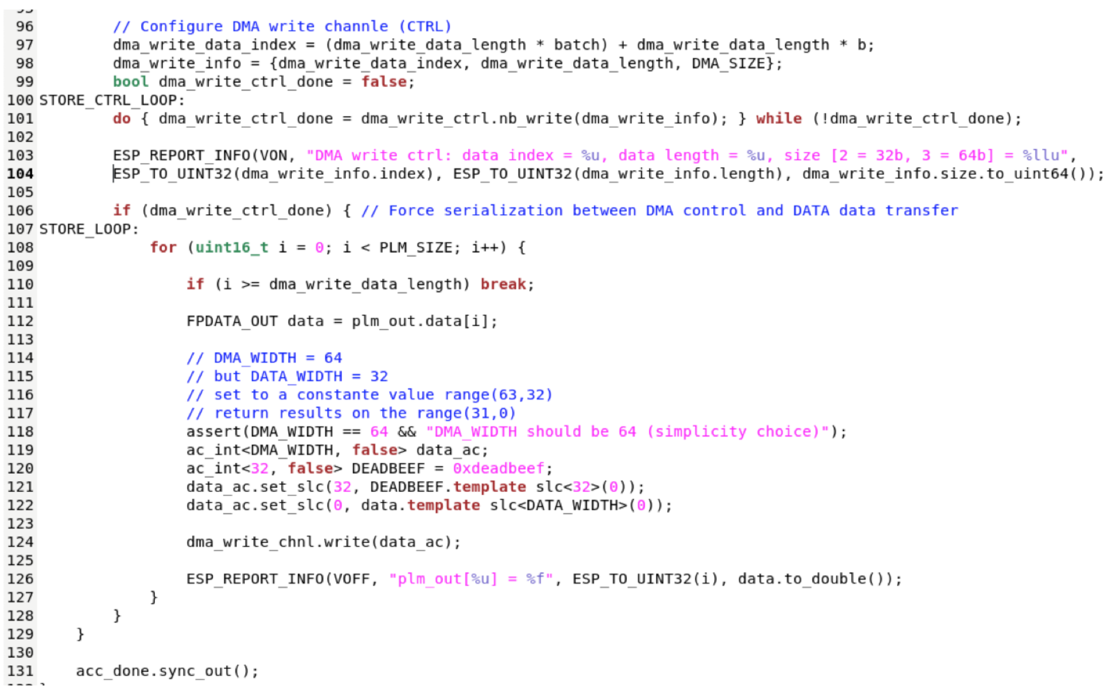
In order to adapt the "Load" phase to our Depthwise accelerator, we need to consider that we have two input memories, one for the ifmap and the other for the kernel, so we need to divide this phase in two "sub-phases". In the first "sub-phase" we need to read data coming from the DMA channel and save them into the ifmap PLM. Then, when we have read an amount of data equal to the maximum size of the ifmap memory, we can move to the second "sub-phase" and complete the "Load" phase saving all the remaining data into the kernel PLM. It is important to notice that this order is only an arbitrary choice coming from the memory definition in the testbench. We can see this modification in Fig. 4.33.



```
        // Configure DMA read channel (CTRL)
        dma_read_data_index = (dma_read_data_length);
        dma_read_info = {dma_read_data_index, dma_read_data_length, DMA_SIZE};
        bool dma_read_ctrl_done = false;
LOAD_CTRL_LOOP:
        do { dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info); } while (!dma_read_ctrl_done);

        ESP_REPORT_INFO(VON, "DMA read ctrl: data index = %u, data length = %u, size [1 = 16b, 2 = 32b, 3 = 64b] = %llu",
        ESP_TO_UINT32(dma_read_info.index), ESP_TO_UINT32(dma_read_info.length), dma_read_info.size.to_uint64());

        if (dma_read_ctrl_done) { // Force serialization between DMA control and DATA data transfer
LOAD_LOOP:
            for (uint16_t i = 0; i < PLM_SIZE+KERNEL_PLM_SIZE; i++) {

                if (i == dma_read_data_length) break;

                assert(DMA_WIDTH == 64 && "DMA_WIDTH should be 64 (simplicity choice)");
                ac_int<DATA_WIDTH, false> data_ac;
#ifndef __SYNTHESIS__
                while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data ready
#endif
                data_ac = dma_read_chnl.read().template slc<DATA_WIDTH>(0);

                FPDATA_IN data;
                data.set_slc(0, data_ac);

  if (i < PLM_SIZE) {

   plm_in.data[i] = data;

                 ESP_REPORT_INFO(VOFF, "plm_in[%u] = %d", ESP_TO_UINT32(i), data.to_int());

  }

  else {

   plm_kernel.data[i-PLM_SIZE] = data;

                 ESP_REPORT_INFO(VOFF, "plm_kernel[%u] = %d", ESP_TO_UINT32(i), data.to_int());

  }


            }
        }
```

Figure 4.33: depthwise.cpp: Load

After the "Load" phase we have the "Compute" phase that is where we have the actual accelerator algorithm that we want to accelerate. In our case we have substituted the Softmax computation with the Depthwise Convolution algorithm. Finally, as we did for the "Load" phase, we need to set the DMA variables for the "Store" phase. In Fig. 4.34 we can see the entire "Store" phase for the Softmax accelerator. This phase does not require any modification because we simply write our accelerator outputs in the DMA channel. It is important to notice that, since the DMA width is assumed to be 64 bits and the data width is 32 bits, when we write something in the DMA channel we need to fill the missing most significant 32 bits with a fixed template. In the Softmax example we can see that the word "0xdeadbeef" is used for convenience. In our Depthwise accelerator we have followed the same approach.

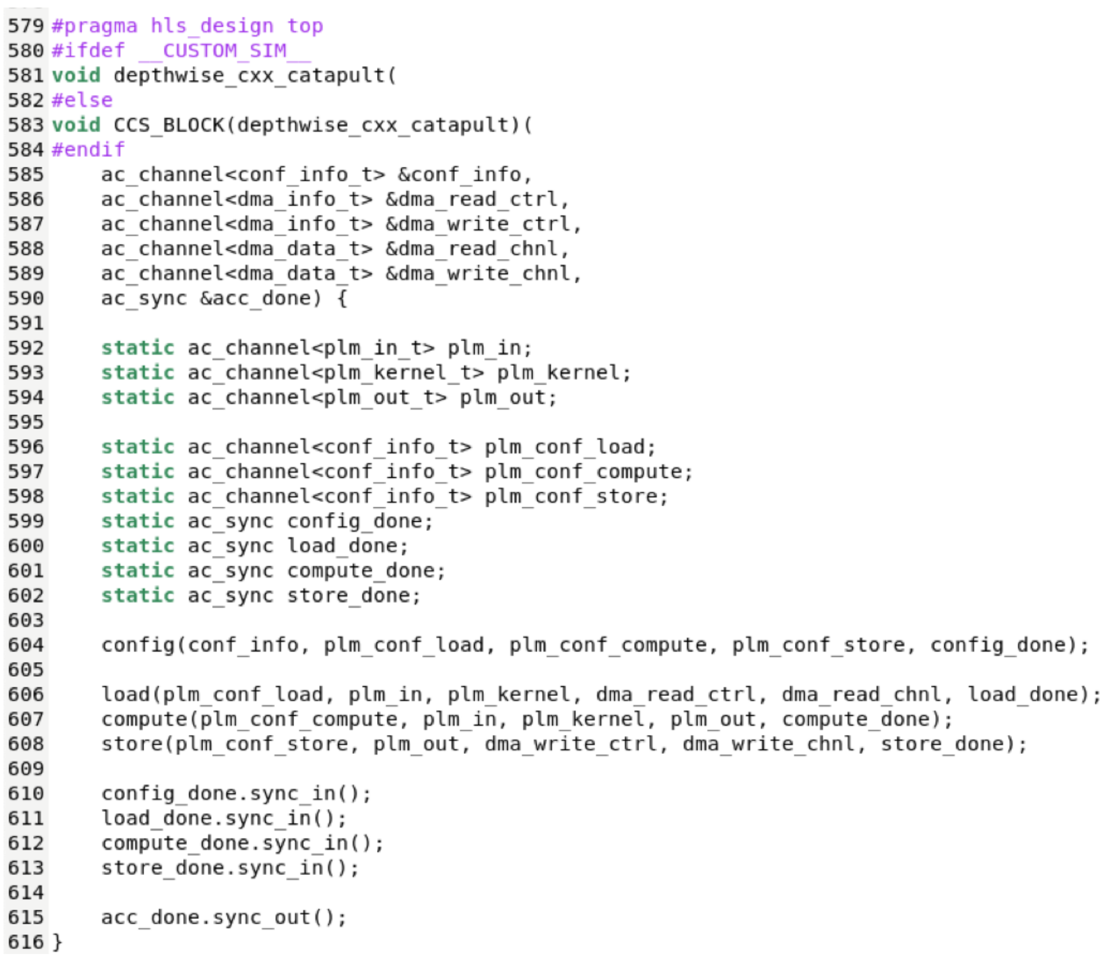


```
        // Configure DMA write channle (CTRL)
        dma_write_data_index = (dma_write_data_length * batch) + dma_write_data_length * b;
        dma_write_info = {dma_write_data_index, dma_write_data_length, DMA_SIZE};
        bool dma_write_ctrl_done = false;
STORE_CTRL_LOOP:
        do { dma_write_ctrl_done = dma_write_ctrl.nb_write(dma_write_info); } while (!dma_write_ctrl_done);

        ESP_REPORT_INFO(VON, "DMA write ctrl: data index = %u, data length = %u, size [2 = 32b, 3 = 64b] = %llu",
        ESP_TO_UINT32(dma_write_info.index), ESP_TO_UINT32(dma_write_info.length), dma_write_info.size.to_uint64());

        if (dma_write_ctrl_done) { // Force serialization between DMA control and DATA data transfer
STORE_LOOP:
            for (uint16_t i = 0; i < PLM_SIZE; i++) {

                if (i >= dma_write_data_length) break;

                FPDATA_OUT data = plm_out.data[i];

                // DMA_WIDTH = 64
                // but DATA_WIDTH = 32
                // set to a constante value range(63,32)
                // return results on the range(31,0)
                assert(DMA_WIDTH == 64 && "DMA_WIDTH should be 64 (simplicity choice)");
                ac_int<DMA_WIDTH, false> data_ac;
                ac_int<32, false> DEADBEEF = 0xdeadbeef;
                data_ac.set_slc(32, DEADBEEF.template slc<32>(0));
                data_ac.set_slc(0, data.template slc<DATA_WIDTH>(0));

                dma_write_chnl.write(data_ac);

                ESP_REPORT_INFO(VOFF, "plm_out[%u] = %f", ESP_TO_UINT32(i), data.to_double());
            }
        }
    }

    acc_done.sync_out();
```

Figure 4.34: softmax.cpp: Store

softmax.cpp/depthwise.cpp (hier)

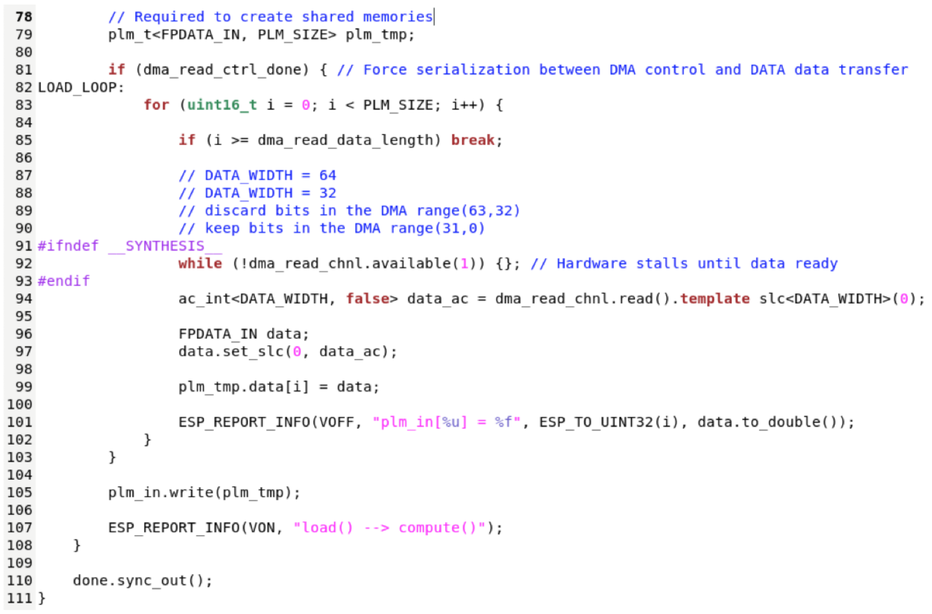
The hierarchical architecture source code file is very similar to the basic architecture one. The only difference is that the "Configure", "Load", "Compute" and "Store" phases now are different C++ functions that are called by the main (top) function in this order as shown in Fig. 4.35.



```
#pragma hls_design top
#ifdef __CUSTOM_SIM__
void depthwise_cxx_catapult(
#else
void CCS_BLOCK(depthwise_cxx_catapult)(
#endif
    ac_channel<conf_info_t> &conf_info,
    ac_channel<dma_info_t> &dma_read_ctrl,
    ac_channel<dma_info_t> &dma_write_ctrl,
    ac_channel<dma_data_t> &dma_read_chnl,
    ac_channel<dma_data_t> &dma_write_chnl,
    ac_sync &acc_done) {

    static ac_channel<plm_in_t> plm_in;
    static ac_channel<plm_kernel_t> plm_kernel;
    static ac_channel<plm_out_t> plm_out;

    static ac_channel<conf_info_t> plm_conf_load;
    static ac_channel<conf_info_t> plm_conf_compute;
    static ac_channel<conf_info_t> plm_conf_store;
    static ac_sync config_done;
    static ac_sync load_done;
    static ac_sync compute_done;
    static ac_sync store_done;

    config(conf_info, plm_conf_load, plm_conf_compute, plm_conf_store, config_done);

    load(plm_conf_load, plm_in, plm_kernel, dma_read_ctrl, dma_read_chnl, load_done);
    compute(plm_conf_compute, plm_in, plm_kernel, plm_out, compute_done);
    store(plm_conf_store, plm_out, dma_write_ctrl, dma_write_chnl, store_done);

    config_done.sync_in();
    load_done.sync_in();
    compute_done.sync_in();
    store_done.sync_in();

    acc_done.sync_out();
}
```

Figure 4.35: depthwise.cpp: hier top function

These functions are all synchronized using a dedicated *ac_sync* done signal that triggers the next phase when the previous one is completed. An important thing that we have to notice is that, since the PLMs are shared among all the phases, in order to avoid unwanted inferred memories, it is a good practice in Catapult HLS to define temporary memories inside each function and then write their content in the actual PLM once the phase is finished. We can see this process in Fig. 4.36 at line 79 and 105 for the load phase.



```
        // Required to create shared memories
        plm_t<FPDATA_IN, PLM_SIZE> plm_tmp;

        if (dma_read_ctrl_done) { // Force serialization between DMA control and DATA data transfer
LOAD_LOOP:
            for (uint16_t i = 0; i < PLM_SIZE; i++) {

                if (i >= dma_read_data_length) break;

                // DATA_WIDTH = 64
                // DATA_WIDTH = 32
                // discard bits in the DMA range(63,32)
                // keep bits in the DMA range(31,0)
#ifndef __SYNTHESIS__
                while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data ready
#endif
                ac_int<DATA_WIDTH, false> data_ac = dma_read_chnl.read().template slc<DATA_WIDTH>(0);

                FPDATA_IN data;
                data.set_slc(0, data_ac);

                plm_tmp.data[i] = data;

                ESP_REPORT_INFO(VOFF, "plm_in[%u] = %f", ESP_TO_UINT32(i), data.to_double());
            }
        }

        plm_in.write(plm_tmp);

        ESP_REPORT_INFO(VON, "load() --> compute()");
    }

    done.sync_out();
}
```

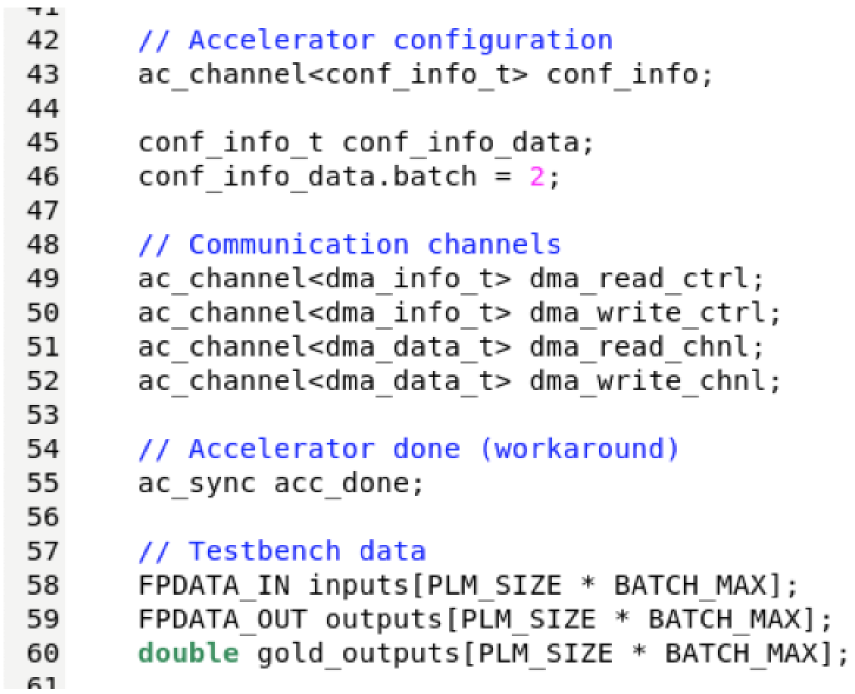
Figure 4.36: softmax.cpp (hier)

4.2.7 *tb* folder

In this folder there is the testbench C++ source code file. This testbench is used during the accelerator RTL simulation in order to provide the inputs to the accelerator and then to check the correctness of the output results with a validation phase.

main.cpp

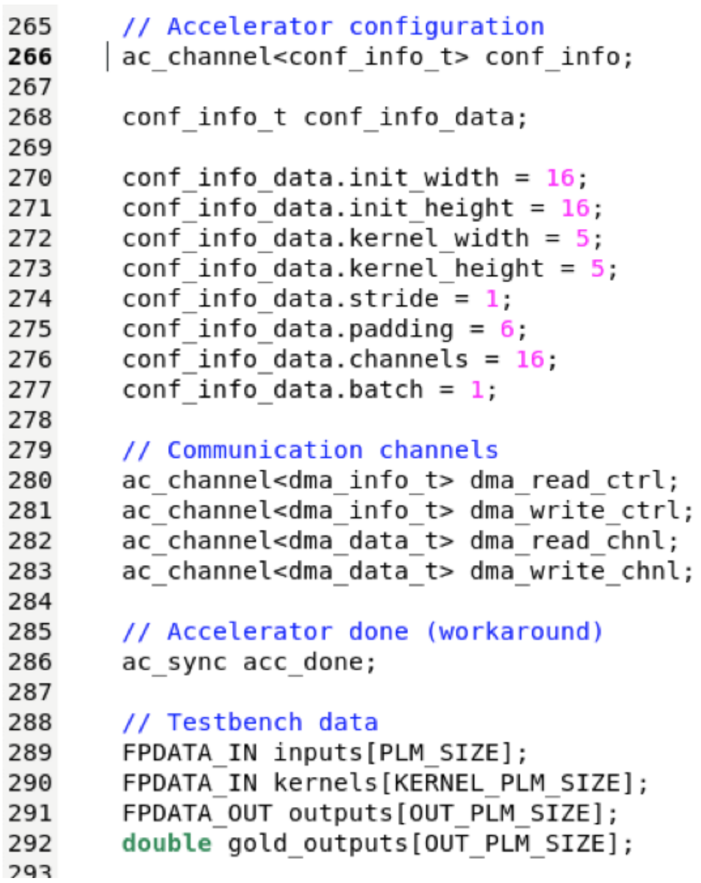
First of all, we have to define the accelerator configuration writing the values that we want to assign to each accelerator parameter. Then we need to create the DMA communication channels in order to be able to provide the inputs to the accelerator and also read the outputs once the accelerator has done. Finally, we need to define the inputs and outputs arrays together with the golden outputs array. The golden outputs will be compared with the accelerator's outputs during the validation phase.
In Fig. 4.37 we can see this first configuration phase for the Softmax accelerator.



```
    // Accelerator configuration
    ac_channel<conf_info_t> conf_info;

    conf_info_t conf_info_data;
    conf_info_data.batch = 2;

    // Communication channels
    ac_channel<dma_info_t> dma_read_ctrl;
    ac_channel<dma_info_t> dma_write_ctrl;
    ac_channel<dma_data_t> dma_read_chnl;
    ac_channel<dma_data_t> dma_write_chnl;

    // Accelerator done (workaround)
    ac_sync acc_done;

    // Testbench data
    FPDATA_IN inputs[PLM_SIZE * BATCH_MAX];
    FPDATA_OUT outputs[PLM_SIZE * BATCH_MAX];
    double gold_outputs[PLM_SIZE * BATCH_MAX];
```

Figure 4.37: softmax main.cpp (Configuration)

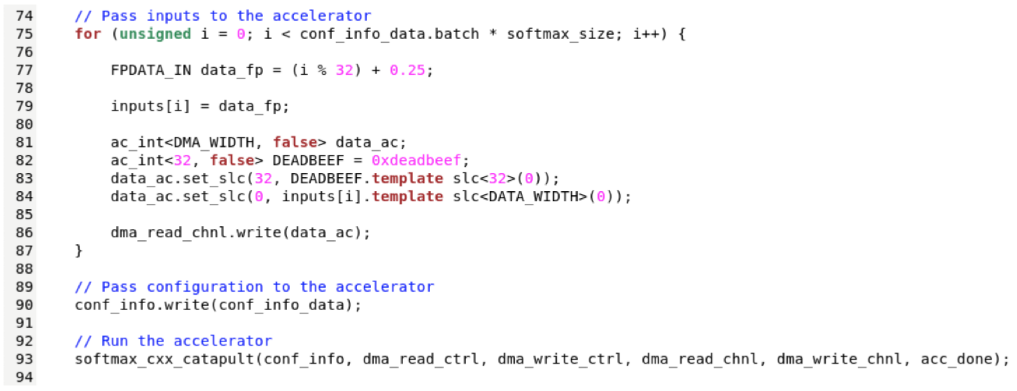
This section must be modified with our specific accelerator parameters and with the required arrays that will be used to fill the accelerator PLMs. In Fig. 4.38 we can see the modifications that we have done in order to configure our Depthwise accelerator. We have inserted all the Convolution parameters and we have added an inputs array that contains the kernels.



```
    // Accelerator configuration
    ac_channel<conf_info_t> conf_info;

    conf_info_t conf_info_data;

    conf_info_data.init_width = 16;
    conf_info_data.init_height = 16;
    conf_info_data.kernel_width = 5;
    conf_info_data.kernel_height = 5;
    conf_info_data.stride = 1;
    conf_info_data.padding = 6;
    conf_info_data.channels = 16;
    conf_info_data.batch = 1;

    // Communication channels
    ac_channel<dma_info_t> dma_read_ctrl;
    ac_channel<dma_info_t> dma_write_ctrl;
    ac_channel<dma_data_t> dma_read_chnl;
    ac_channel<dma_data_t> dma_write_chnl;

    // Accelerator done (workaround)
    ac_sync acc_done;

    // Testbench data
    FPDATA_IN inputs[PLM_SIZE];
    FPDATA_IN kernels[KERNEL_PLM_SIZE];
    FPDATA_OUT outputs[OUT_PLM_SIZE];
    double gold_outputs[OUT_PLM_SIZE];
```

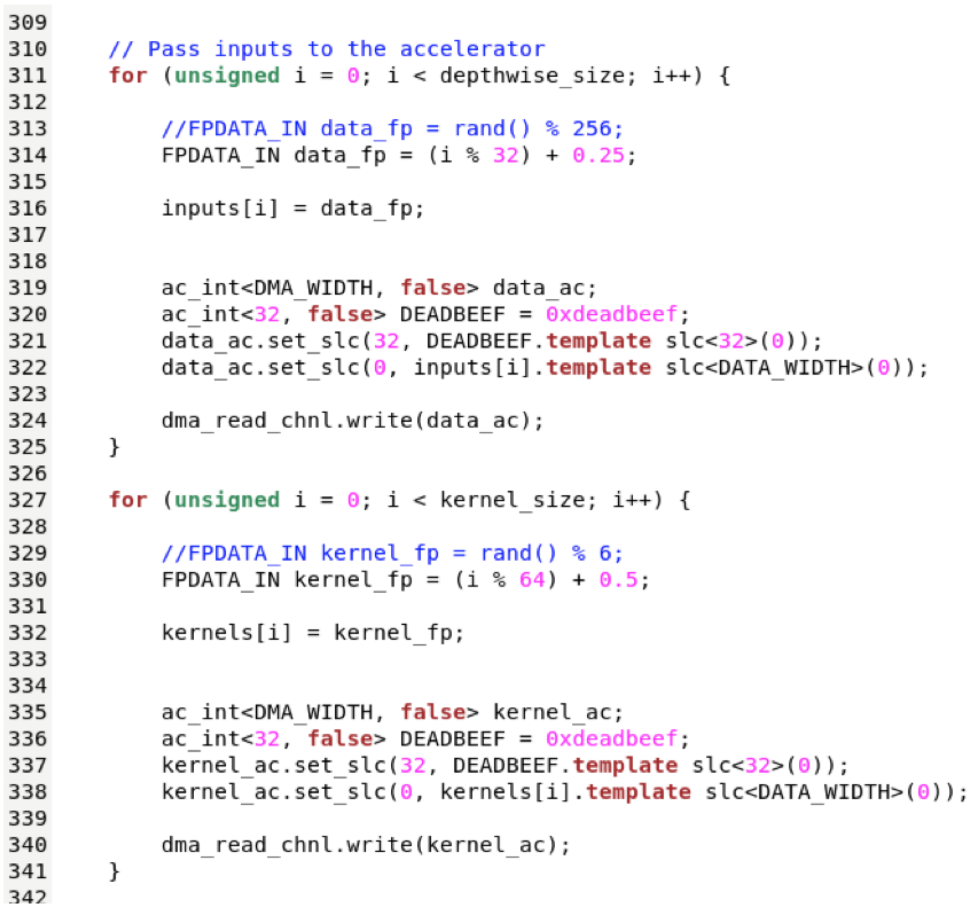
Figure 4.38: depthwise main.cpp (Configuration)

Then we have to pass these arrays to the accelerator. To do that, we need to write into the *dma_read_chnl* the data contained inside the inputs memory using the *ac_channel write()* function. It is important to notice that also in this case we need to pay attention to the DMA width and data width. In fact, if we consider that the DMA width is equal to 64 bits and the data width is equal to 32 bits, writing these data into the DMA channel we need to fill the most significant 32 bits with a template. As we did also in the accelerator C++ source code file, we can use as template the word "0xdeadbeef" that will fill the missing 32 bits. The next step is to pass the parameters configuration to the accelerator. This is a simpler process because we only need to use the *ac_channel write()* function using the configuration struct *conf_info_data* as argument. After that, we can finally run the accelerator simply calling the accelerator top module function using as arguments the DMA interfaces. In Fig. 4.39 we can see all these steps for the Softmax accelerator.



```
// Pass inputs to the accelerator
for (unsigned i = 0; i < conf_info_data.batch * softmax_size; i++) {

    FPDATA_IN data_fp = (i % 32) + 0.25;

    inputs[i] = data_fp;

    ac_int<DMA_WIDTH, false> data_ac;
    ac_int<32, false> DEADBEEF = 0xdeadbeef;
    data_ac.set_slc(32, DEADBEEF.template slc<32>(0));
    data_ac.set_slc(0, inputs[i].template slc<DATA_WIDTH>(0));

    dma_read_chnl.write(data_ac);
}

// Pass configuration to the accelerator
conf_info.write(conf_info_data);

// Run the accelerator
softmax_cxx_catapult(conf_info, dma_read_ctrl, dma_write_ctrl, dma_read_chnl, dma_write_chnl, acc_done);

```

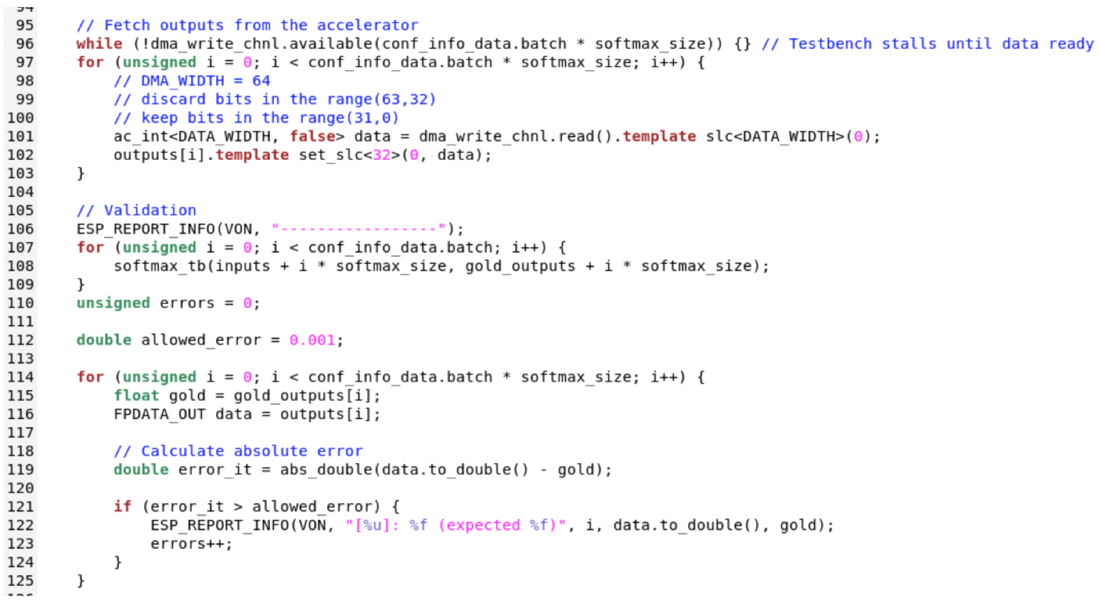
Figure 4.39: softmax main.cpp (Inputs writing)



```

// Pass inputs to the accelerator
for (unsigned i = 0; i < depthwise_size; i++) {

    //FPDATA_IN data_fp = rand() % 256;
    FPDATA_IN data_fp = (i % 32) + 0.25;

    inputs[i] = data_fp;


    ac_int<DMA_WIDTH, false> data_ac;
    ac_int<32, false> DEADBEEF = 0xdeadbeef;
    data_ac.set_slc(32, DEADBEEF.template slc<32>(0));
    data_ac.set_slc(0, inputs[i].template slc<DATA_WIDTH>(0));

    dma_read_chnl.write(data_ac);
}

for (unsigned i = 0; i < kernel_size; i++) {

    //FPDATA_IN kernel_fp = rand() % 6;
    FPDATA_IN kernel_fp = (i % 64) + 0.5;

    kernels[i] = kernel_fp;


    ac_int<DMA_WIDTH, false> kernel_ac;
    ac_int<32, false> DEADBEEF = 0xdeadbeef;
    kernel_ac.set_slc(32, DEADBEEF.template slc<32>(0));
    kernel_ac.set_slc(0, kernels[i].template slc<DATA_WIDTH>(0));

    dma_read_chnl.write(kernel_ac);
}

```

Figure 4.40: depthwise main.cpp (Inputs writing)

In order to pass the stimuli to our Depthwise accelerator, we write on the *dma_read_chnl* the ifmap array, and then the kernel array. We can see these two writing operations in Fig. 4.40. The last thing that we have to do is to fetch the accelerator outputs and compare them with the golden outputs obtained from the testbench function. The only modification that is required for this validation phase is to generate the golden output inside the testbench. In our case we replaced the Softmax function with the Depthwise one. Fig. 4.41 reports the validation phase for the Softmax accelerator.

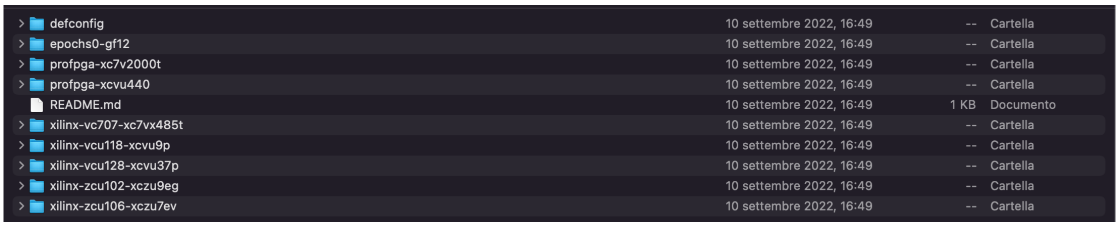


```
    // Fetch outputs from the accelerator
    while (!dma_write_chnl.available(conf_info_data.batch * softmax_size)) {} // Testbench stalls until data ready
    for (unsigned i = 0; i < conf_info_data.batch * softmax_size; i++) {
        // DMA_WIDTH = 64
        // discard bits in the range(63,32)
        // keep bits in the range(31,0)
        ac_int<DATA_WIDTH, false> data = dma_write_chnl.read().template slc<DATA_WIDTH>(0);
        outputs[i].template set_slc<32>(0, data);
    }

    // Validation
    ESP_REPORT_INFO(VON, "-----------------");
    for (unsigned i = 0; i < conf_info_data.batch; i++) {
        softmax_tb(inputs + i * softmax_size, gold_outputs + i * softmax_size);
    }
    unsigned errors = 0;

    double allowed_error = 0.001;

    for (unsigned i = 0; i < conf_info_data.batch * softmax_size; i++) {
        float gold = gold_outputs[i];
        FPDATA_OUT data = outputs[i];

        // Calculate absolute error
        double error_it = abs_double(data.to_double() - gold);

        if (error_it > allowed_error) {
            ESP_REPORT_INFO(VON, "[%u]: %f (expected %f)", i, data.to_double(), gold);
            errors++;
        }
    }
```

Figure 4.41: softmax.cpp (Validation)

4.3 Accelerator Synthesis & Simulation

Now that the design of the accelerator is completed, we can synthesize and simulate it running the build_prj_top.tcl script In order to run our script we need to go back in the *esp* root folder and move in the *socs* directory. Here there are several folders, one for each supported FPGA. In our case, we need to go into the *profpga-xc7v2000t* folder. We can see all the directories inside the *socs* folder in Fig. 4.42.



```
> defconfig                  10 settembre 2022, 16:49    --    Cartella
> epochs0-gf12               10 settembre 2022, 16:49    --    Cartella
> profpga-xc7v2000t          10 settembre 2022, 16:49    --    Cartella
> profpga-xcvu440            10 settembre 2022, 16:49    --    Cartella
  README.md                  10 settembre 2022, 16:49  1 KB    Documento
> xilinx-vc707-xc7vx485t     10 settembre 2022, 16:49    --    Cartella
> xilinx-vcu118-xcvu9p       10 settembre 2022, 16:49    --    Cartella
> xilinx-vcu128-xcvu37p      10 settembre 2022, 16:49    --    Cartella
> xilinx-zcu102-xczu9eg      10 settembre 2022, 16:49    --    Cartella
> xilinx-zcu106-xczu7ev      10 settembre 2022, 16:49    --    Cartella
```

Figure 4.42: socs folder

Now from the *profpga-xc7v2000t* folder we can run this command on the shell to execute the HLS script:

$$DMA\_WIDTH{=}64\ make\ depthwise\_cxx\_catapult\text{-}hls \tag{4.1}$$

After the synthesis process and the C co-simulation, Questasim shows up. We only have to run the simulation and check if the validation phase is passed correctly with 0 errors. In addition, we can also see the waveforms in order to better understand the behavior of the accelerator.

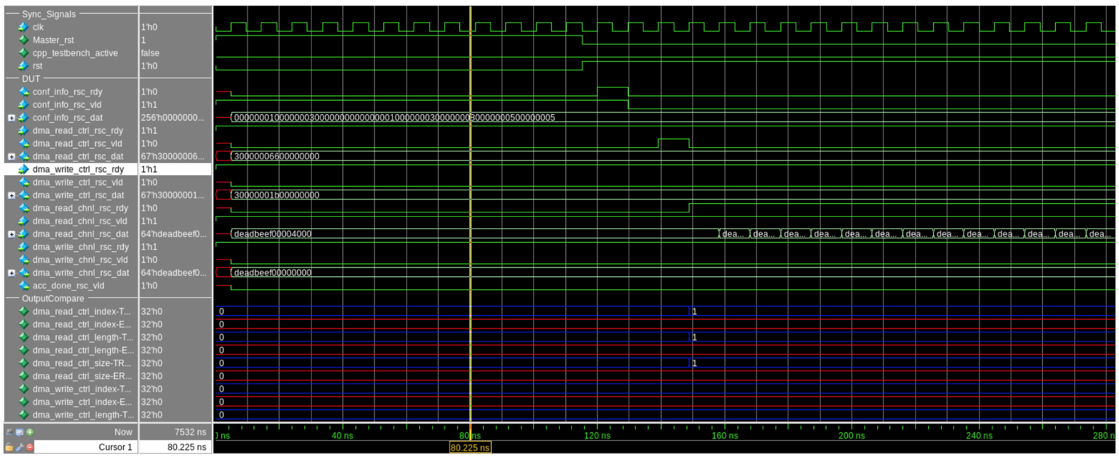


Figure 4.43: depthwise waveforms (Configure & Load)

In Fig. 4.43 we can see how the "Configure" phase and "Load" phase are done. When the reset is de-asserted, the *conf_info_rsc_rdy* signal goes up to 1 and, together with the ;extitconf_info_rsc_vld tells the accelerator to read the configuration parameters. Then, after setting the DMA variables, the *dma_read_ctrl_rsc_vld* goes up to 1 and in the next clock cycle also the *dma_read_chnl_rsc_rdy* goes from 0 to 1: This will start the "Load" phase.

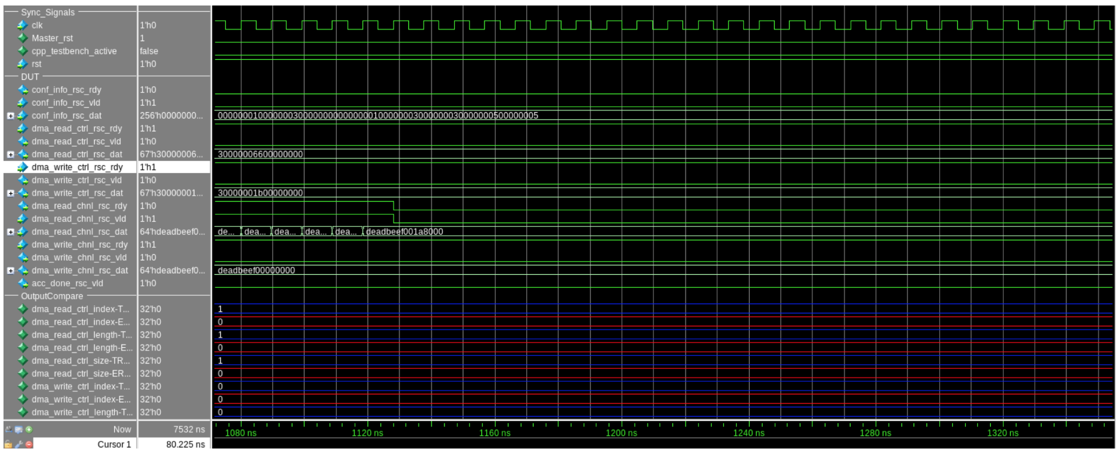


Figure 4.44: depthwise waveforms (Compute)

From Fig. 4.44 we can see that after reading all the data the two signals *dma_read_chnl_rsc_rdy* and *dma_read_chnl_rsc_vld* go from 1 to 0, then the "Compute" phase will start.

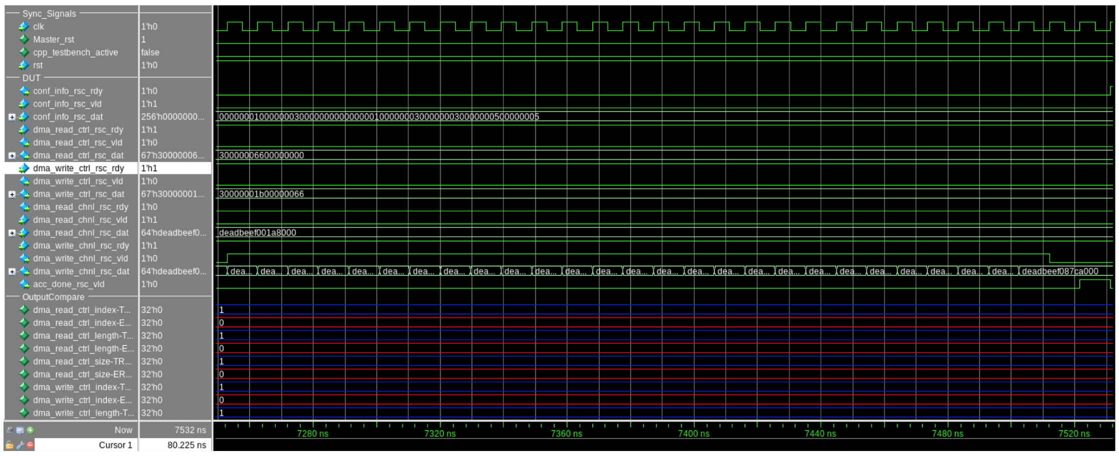


Figure 4.45: depthwise waveforms (Store)

When the "Compute" phase ends, as we can see in Fig. 4.45, the validation signal *dma_write_chnl_rsc_vld* goes up to 1 starting the "Store" phase. Then, when all the writings are completed, the "Store" ends with the *dma_write_chnl_rsc_vld* signal that goes back to 0. Finally, at the next clock cycle the done signal *acc_done_rsc_vld* goes up to 1 and after that the simulation ends.

Chapter 5

Create a SoC in ESP and implement it on FPGA

The main purpose of the ESP accelerator design flow is the creation of a large set of IP components that can be selected to build a complete SoC leveraging the modularity provided by the platform [9]. The ESP SoC design flow is totally based on a GUI that helps the designer with an interactive design process. In Fig. 5.1 we can see how this GUI looks like and which are the main steps for the accelerator design flow and the SoC design flow.

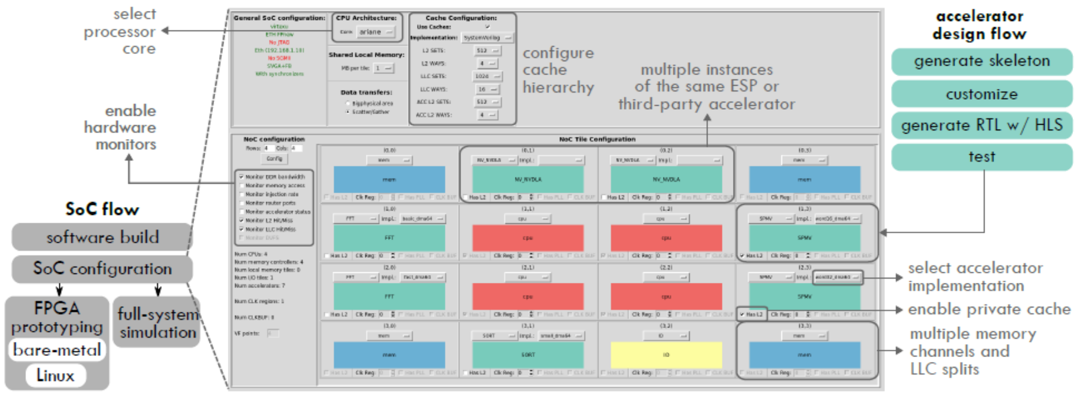


Figure 5.1: ESP GUI (Image taken from [9])

Thanks to this GUI it is possible to configure the SoC choosing [9]:

- the number, the position and the type of tiles.
- the desired accelerator to insert in each accelerator tile

- the desired processor core
- the cache hierarchy configuration
- the clock domain for each tile
- the desired system monitors

In order to complete the SoC configuration process we need to click on the *Generate SoC config* push-button within the GUI, after that, ESP generates a configuration file that is used during the simulation process to generate [9]:

- the RTL sockets
- the NoC routing tables
- the system memory mapping
- the device tree for the target processor
- the configuration parameters for all the proxy components
- the software header files

Finally, once the SoC is completed, the designer can run a full SoC RTL simulation with a bare-metal program running on the soft-core processor. In addition to that, using a specific *make* target, it is possible to generate the bitstream for one of the supported FPGA boards, that at the moment are [9]:

1. Xilinx XCU128
2. Xilinx VCU118
3. Xilinx VC707
4. Profpga XCVU440
5. Profpga XC7V2000T

In particular, if we monitor the FPGA with the UART interface we can also run the bare-metal program on the FPGA. ESP gives also the possibility to run Linux application on the selected soft-core processor, but this will not be analyzed in this thesis. It is possible to find more information in the reference document [9].

5.1 Write the Bare-metal application

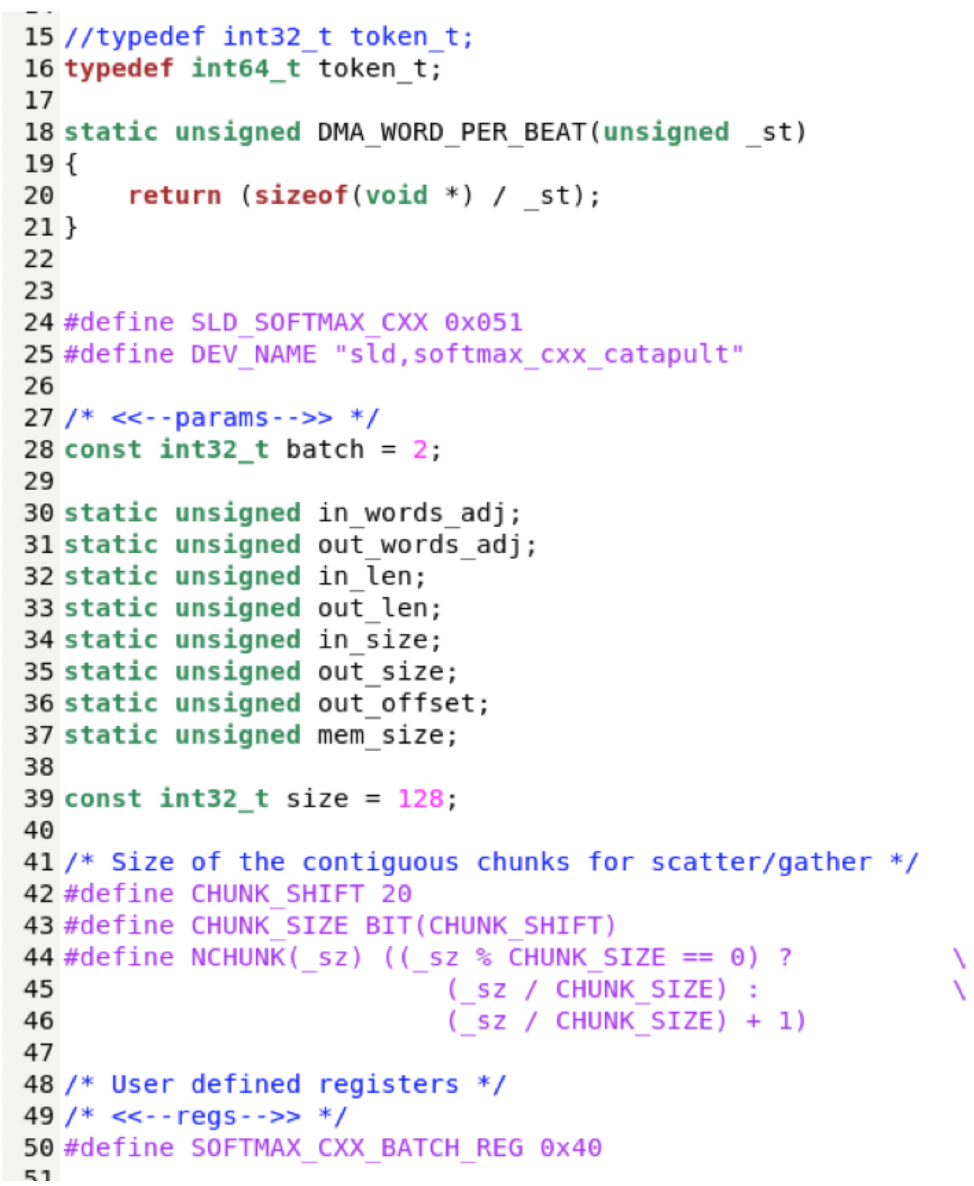
In the previous chapter we have analyzed the content of the *hw* folder inside the Catapult HLS accelerators main directory for the Softmax accelerator example. Then we have shown how to modify all the files in order to adapt this sample to a generic accelerator, in our case, the Depthwise accelerator. Now we analyze what is inside the *sw* folder. Inside the *sw* folder there are other two different folders: the *baremetal* folder which contains the bare-metal application and the *Linux* folder which contains all the files required to compile Linux. Since we are not dealing with Linux, the next paragraph is dedicated to the *baremetal* folder only.

5.1.1 *baremetal* folder

Inside this folder we have the actual bare-metal application C source code file (.c) that we can use as a software testbench for the SoC in order to see if our new accelerator is working correctly inside the complete system.

softmax.c/depthwise.c

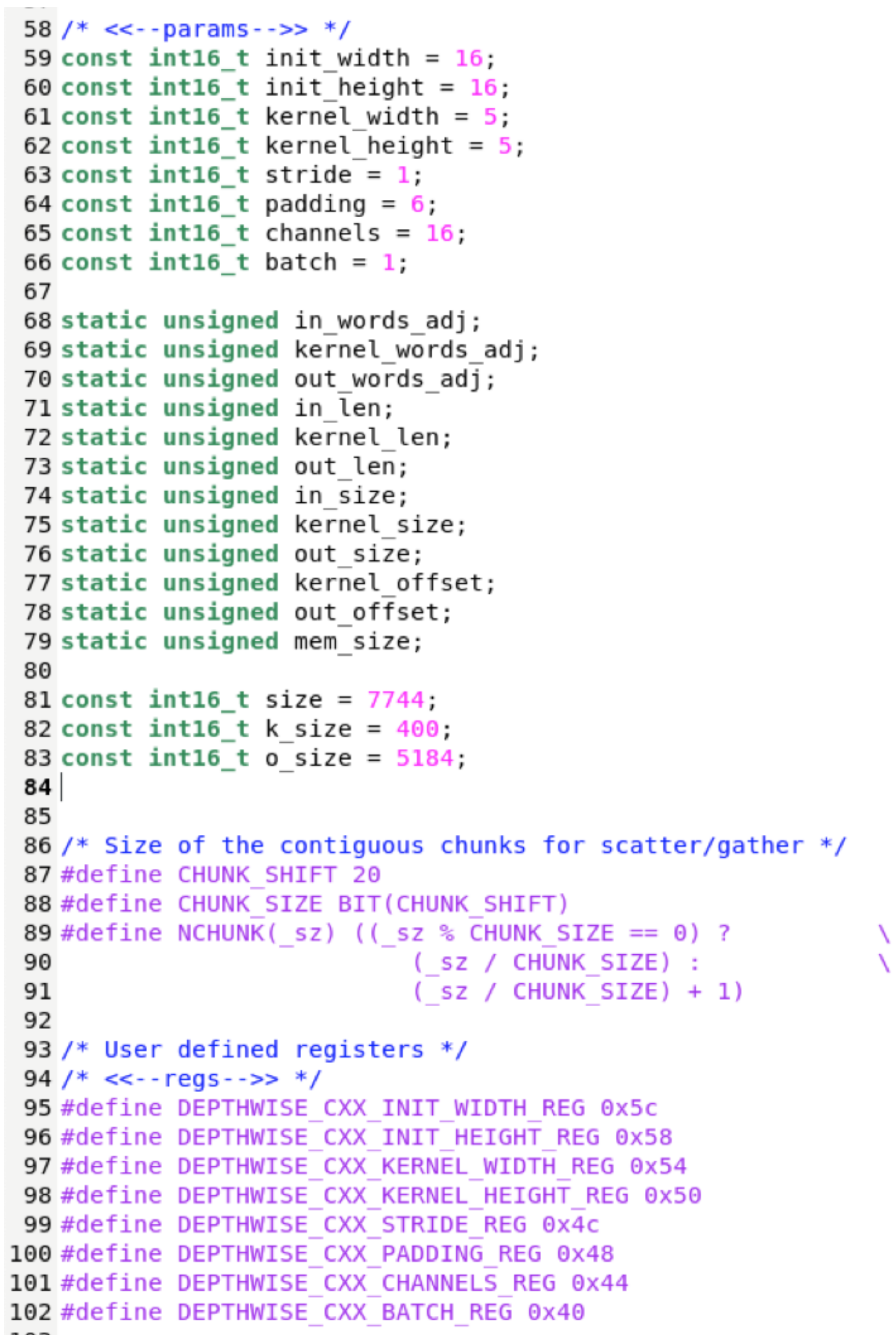
First of all, as we can see in Fig. 5.2, we define a new datatype called *token_t* that will be the datatype used inside the memories and during the DMA transfers. In this case this is an *int64_t* because we have chosen a 64 bits DMA. Then we need to define the accelerator name and its unique ID that we have provided in the *xml* file inside the *hw* folder of the accelerator. The most important part of this section is the definition of all the accelerator parameters, the PLMs dimensions and the addresses of the user defined registers. These registers will be the physical location where the accelerator will read its parameters. As we can see from the case of the Softmax accelerator, since we need only the batch parameter, we will use only one register at the address $0x40$. It is important to notice that we have to choose this addresses carefully because: we do not want to write on registers used by the system for other internal parameters, and two consecutive addresses are separated by 16 bits. For example, if we want to add a parameter into the Softmax accelerator, we need to assign it to the register at the $0x44$ address. The complete address map of I/O registers defined by ESP is available at the following resource [19].



```c
//typedef int32_t token_t;
typedef int64_t token_t;

static unsigned DMA_WORD_PER_BEAT(unsigned _st)
{
    return (sizeof(void *) / _st);
}


#define SLD_SOFTMAX_CXX 0x051
#define DEV_NAME "sld,softmax_cxx_catapult"

/* <<--params-->> */
const int32_t batch = 2;

static unsigned in_words_adj;
static unsigned out_words_adj;
static unsigned in_len;
static unsigned out_len;
static unsigned in_size;
static unsigned out_size;
static unsigned out_offset;
static unsigned mem_size;

const int32_t size = 128;

/* Size of the contiguous chunks for scatter/gather */
#define CHUNK_SHIFT 20
#define CHUNK_SIZE BIT(CHUNK_SHIFT)
#define NCHUNK(_sz) ((_sz % CHUNK_SIZE == 0) ?          \
                        (_sz / CHUNK_SIZE) :            \
                        (_sz / CHUNK_SIZE) + 1)

/* User defined registers */
/* <<--regs-->> */
#define SOFTMAX_CXX_BATCH_REG 0x40
```

Figure 5.2: softmax.c (Parameters & Registers definition)

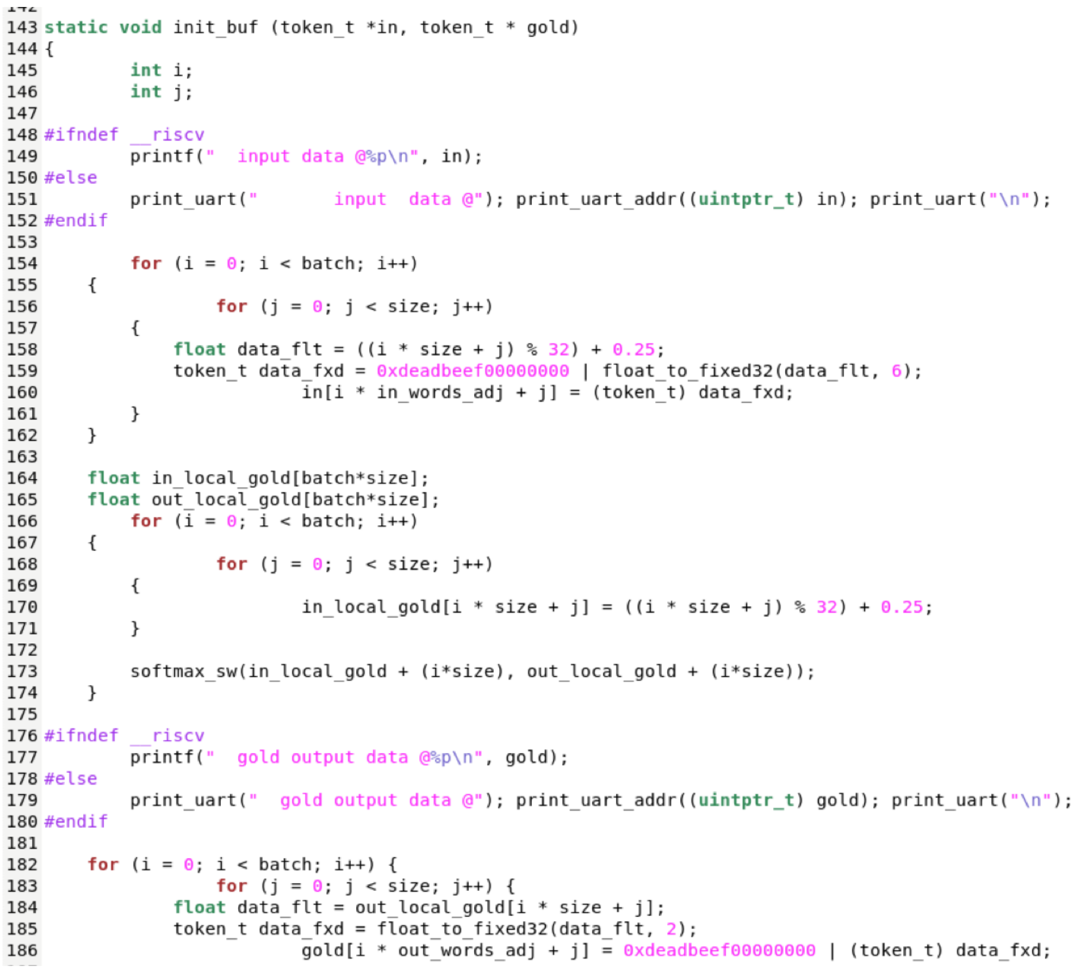
For our Depthwise accelerator, as shown in Fig. 5.3, we have decided to assign these addresses starting again from $0x40$ and going up to $0x5c$. Another important aspect that we need to take into account is that the parameters must be addressed in the same order as they are defined in the *xml* accelerator file. In our case, since our *INIT_WIDTH_REG* is assigned to the largest address, it must be the last parameter defined in the *xml* file.



```
/* <<--params-->> */
const int16_t init_width = 16;
const int16_t init_height = 16;
const int16_t kernel_width = 5;
const int16_t kernel_height = 5;
const int16_t stride = 1;
const int16_t padding = 6;
const int16_t channels = 16;
const int16_t batch = 1;

static unsigned in_words_adj;
static unsigned kernel_words_adj;
static unsigned out_words_adj;
static unsigned in_len;
static unsigned kernel_len;
static unsigned out_len;
static unsigned in_size;
static unsigned kernel_size;
static unsigned out_size;
static unsigned kernel_offset;
static unsigned out_offset;
static unsigned mem_size;

const int16_t size = 7744;
const int16_t k_size = 400;
const int16_t o_size = 5184;


/* Size of the contiguous chunks for scatter/gather */
#define CHUNK_SHIFT 20
#define CHUNK_SIZE BIT(CHUNK_SHIFT)
#define NCHUNK(_sz) ((_sz % CHUNK_SIZE == 0) ?          \
                        (_sz / CHUNK_SIZE) :            \
                        (_sz / CHUNK_SIZE) + 1)

/* User defined registers */
/* <<--regs-->> */
#define DEPTHWISE_CXX_INIT_WIDTH_REG 0x5c
#define DEPTHWISE_CXX_INIT_HEIGHT_REG 0x58
#define DEPTHWISE_CXX_KERNEL_WIDTH_REG 0x54
#define DEPTHWISE_CXX_KERNEL_HEIGHT_REG 0x50
#define DEPTHWISE_CXX_STRIDE_REG 0x4c
#define DEPTHWISE_CXX_PADDING_REG 0x48
#define DEPTHWISE_CXX_CHANNELS_REG 0x44
#define DEPTHWISE_CXX_BATCH_REG 0x40
```

Figure 5.3: depthwise.c (Parameters & Registers definition)

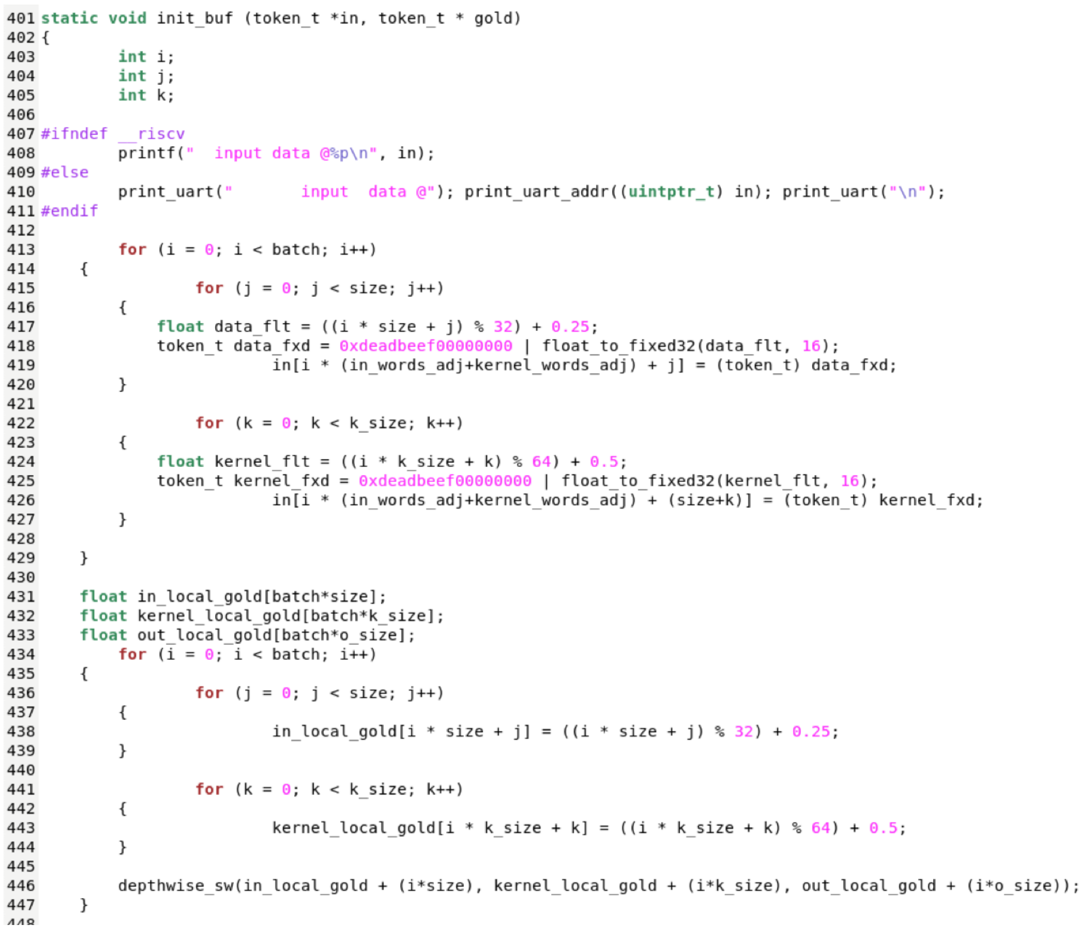
Then we need a function that initializes the input buffers, used later for the accelerator call, and that produces the golden outputs, stored in the golden output memory. This function is called *init_buf* and is reported in Fig. 5.4. The C code that computes the golden outputs starting from the input values can be the same code used for the C testbench discussed in Paragraph 4.2.7.



```
static void init_buf (token_t *in, token_t * gold)
{
        int i;
        int j;

#ifndef __riscv
        printf("  input data @%p\n", in);
#else
        print_uart("       input  data @"); print_uart_addr((uintptr_t) in); print_uart("\n");
#endif

        for (i = 0; i < batch; i++)
    {
                for (j = 0; j < size; j++)
        {
            float data_flt = ((i * size + j) % 32) + 0.25;
            token_t data_fxd = 0xdeadbeef00000000 | float_to_fixed32(data_flt, 6);
                        in[i * in_words_adj + j] = (token_t) data_fxd;
        }
    }

    float in_local_gold[batch*size];
    float out_local_gold[batch*size];
        for (i = 0; i < batch; i++)
    {
                for (j = 0; j < size; j++)
        {
                        in_local_gold[i * size + j] = ((i * size + j) % 32) + 0.25;
        }

        softmax_sw(in_local_gold + (i*size), out_local_gold + (i*size));
    }

#ifndef __riscv
        printf("  gold output data @%p\n", gold);
#else
        print_uart("  gold output data @"); print_uart_addr((uintptr_t) gold); print_uart("\n");
#endif

    for (i = 0; i < batch; i++) {
                for (j = 0; j < size; j++) {
            float data_flt = out_local_gold[i * size + j];
            token_t data_fxd = float_to_fixed32(data_flt, 2);
                        gold[i * out_words_adj + j] = 0xdeadbeef00000000 | (token_t) data_fxd;
```

Figure 5.4: softmax.c (buffer initialization function)

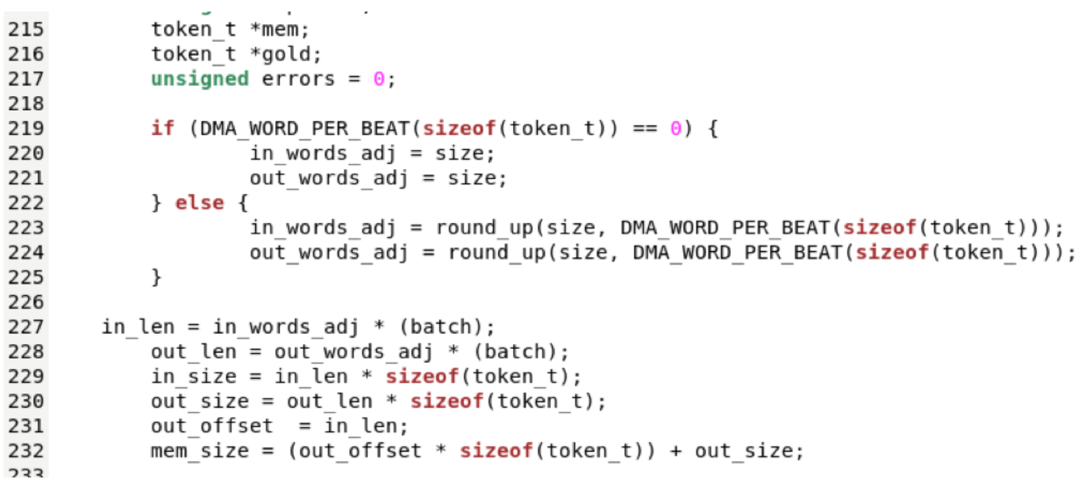
To adapt this function to our Depthwise accelerator, we need to separate the initialization of the ifmap from the initialization of the kernels, considering that both will come from the same input memory. Then, we need to insert the Depthwise algorithm instead of the Softmax one. We can see all these changes in Fig. 5.5.



```
static void init_buf (token_t *in, token_t * gold)
{
        int i;
        int j;
        int k;

#ifndef __riscv
        printf("  input data @%p\n", in);
#else
        print_uart("       input  data @"); print_uart_addr((uintptr_t) in); print_uart("\n");
#endif

        for (i = 0; i < batch; i++)
    {
                for (j = 0; j < size; j++)
        {
            float data_flt = ((i * size + j) % 32) + 0.25;
            token_t data_fxd = 0xdeadbeef00000000 | float_to_fixed32(data_flt, 16);
                        in[i * (in_words_adj+kernel_words_adj) + j] = (token_t) data_fxd;
        }

                for (k = 0; k < k_size; k++)
        {
            float kernel_flt = ((i * k_size + k) % 64) + 0.5;
            token_t kernel_fxd = 0xdeadbeef00000000 | float_to_fixed32(kernel_flt, 16);
                        in[i * (in_words_adj+kernel_words_adj) + (size+k)] = (token_t) kernel_fxd;
        }

    }

    float in_local_gold[batch*size];
    float kernel_local_gold[batch*k_size];
    float out_local_gold[batch*o_size];
        for (i = 0; i < batch; i++)
    {
                for (j = 0; j < size; j++)
        {
                        in_local_gold[i * size + j] = ((i * size + j) % 32) + 0.25;
        }

                for (k = 0; k < k_size; k++)
        {
                        kernel_local_gold[i * k_size + k] = ((i * k_size + k) % 64) + 0.5;
        }

        depthwise_sw(in_local_gold + (i*size), kernel_local_gold + (i*k_size), out_local_gold + (i*o_size));
    }
```

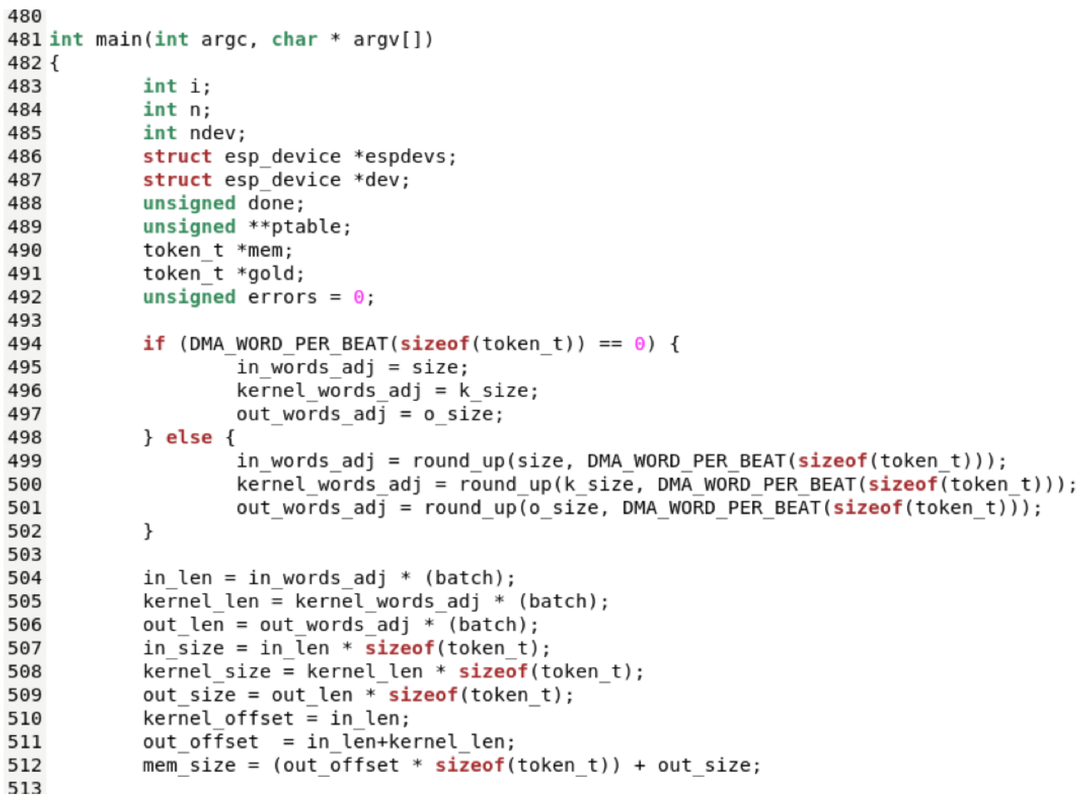
Figure 5.5: depthwise.c (buffer initialization function)

Now we can move into the main function of the bare-metal program. Here, as a first step, we have the definition of the actual memories. In particular we have to notice that, if we use a single memory tile inside our SoC, we have to use only one single memory that will contain both the inputs and the outputs also in this code. For this reason is very important to define in a correct way the size of the memory. In Fig. 5.6 we can see that the memory size for the Softmax accelerator is computed as $in\_size + out\_size$. Another useful variable defined here is the out_offset: this variable acts like an index that point to the cell where the first output is stored.



```c
        token_t *mem;
        token_t *gold;
        unsigned errors = 0;

        if (DMA_WORD_PER_BEAT(sizeof(token_t)) == 0) {
                in_words_adj = size;
                out_words_adj = size;
        } else {
                in_words_adj = round_up(size, DMA_WORD_PER_BEAT(sizeof(token_t)));
                out_words_adj = round_up(size, DMA_WORD_PER_BEAT(sizeof(token_t)));
        }

    in_len = in_words_adj * (batch);
        out_len = out_words_adj * (batch);
        in_size = in_len * sizeof(token_t);
        out_size = out_len * sizeof(token_t);
        out_offset  = in_len;
        mem_size = (out_offset * sizeof(token_t)) + out_size;
```

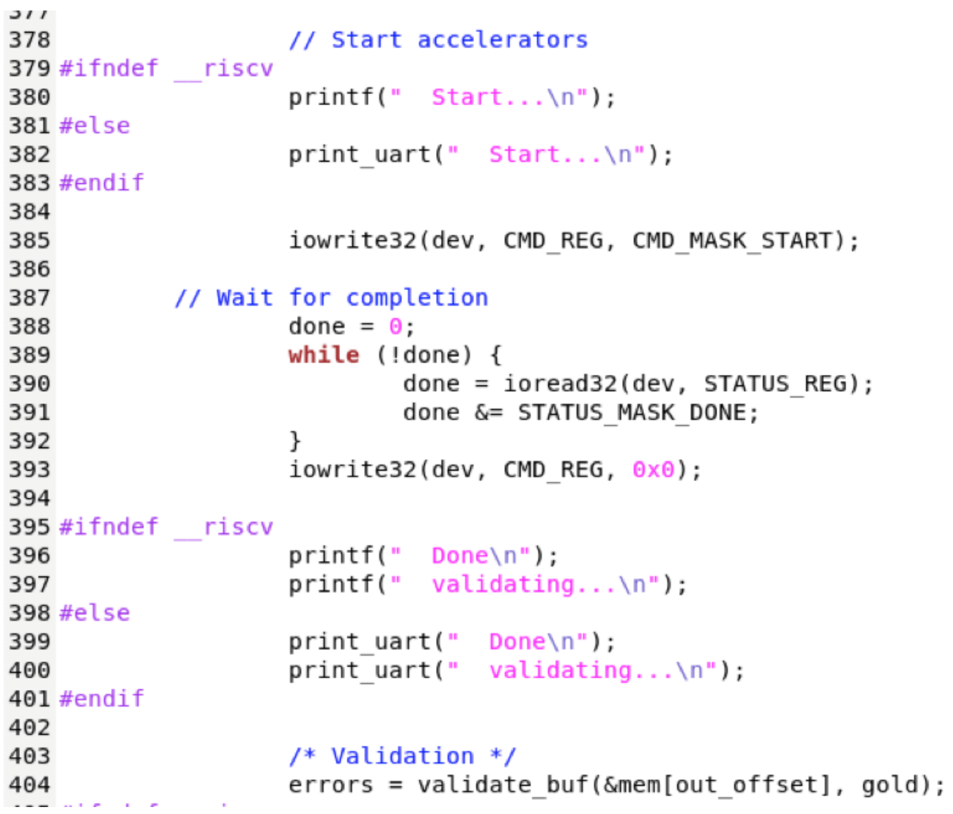
Figure 5.6: softmax.c (main function memories definition)



```c

int main(int argc, char * argv[])
{
        int i;
        int n;
        int ndev;
        struct esp_device *espdevs;
        struct esp_device *dev;
        unsigned done;
        unsigned **ptable;
        token_t *mem;
        token_t *gold;
        unsigned errors = 0;

        if (DMA_WORD_PER_BEAT(sizeof(token_t)) == 0) {
                in_words_adj = size;
                kernel_words_adj = k_size;
                out_words_adj = o_size;
        } else {
                in_words_adj = round_up(size, DMA_WORD_PER_BEAT(sizeof(token_t)));
                kernel_words_adj = round_up(k_size, DMA_WORD_PER_BEAT(sizeof(token_t)));
                out_words_adj = round_up(o_size, DMA_WORD_PER_BEAT(sizeof(token_t)));
        }

        in_len = in_words_adj * (batch);
        kernel_len = kernel_words_adj * (batch);
        out_len = out_words_adj * (batch);
        in_size = in_len * sizeof(token_t);
        kernel_size = kernel_len * sizeof(token_t);
        out_size = out_len * sizeof(token_t);
        kernel_offset = in_len;
        out_offset  = in_len+kernel_len;
        mem_size = (out_offset * sizeof(token_t)) + out_size;
```

Figure 5.7: depthwise.c (main function memories definition)

For our Depthwise accelerator we need to modify these memory sizes taking into account that our memory will be realized by three different sections: the ifmap section, the kernels section and the ofmap section. So our sizes become in_size+$kernel\_size$+out_size. We need also to modify the out_offset variable accordingly. These changes are reported in Fig. 5.7. The last code section that is important to analyze is the section in which we give the start command to the accelerator waiting for its completion and then we perform the validation phase and it is valid both for the Softmax and Depthwise baremetal C codes. This is shown in Fig. 5.8. As we can notice, in order to give the start to the accelerator, we



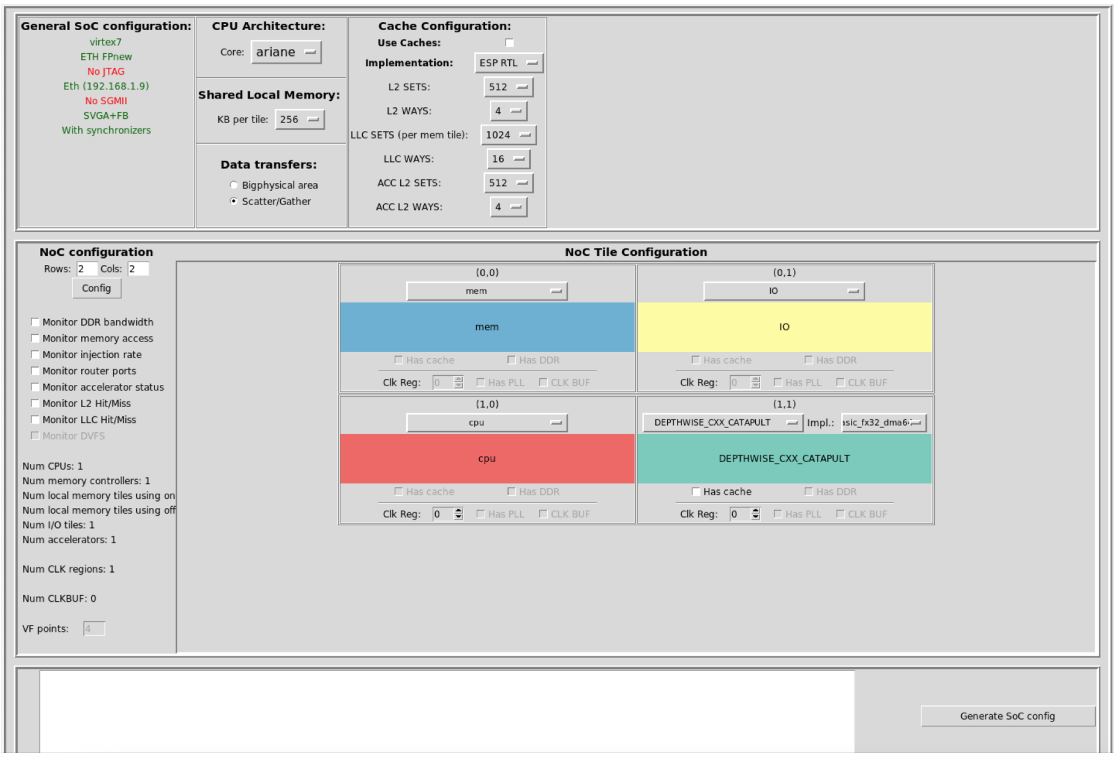
```
        // Start accelerators
#ifndef __riscv
        printf("  Start...\n");
#else
        print_uart("  Start...\n");
#endif

        iowrite32(dev, CMD_REG, CMD_MASK_START);

    // Wait for completion
        done = 0;
        while (!done) {
            done = ioread32(dev, STATUS_REG);
            done &= STATUS_MASK_DONE;
        }
        iowrite32(dev, CMD_REG, 0x0);

#ifndef __riscv
        printf("  Done\n");
        printf("  validating...\n");
#else
        print_uart("  Done\n");
        print_uart("  validating...\n");
#endif

        /* Validation */
        errors = validate_buf(&mem[out_offset], gold);
```

Figure 5.8: softmax.c (accelerator run)

simply write a fixed bit mask called *CMD_MASK_START* using the *iowrite32()* function into a specific memory-mapped register called *CMD_REG*. At this point we poll the register called *STATUS_REG* with the *ioread32()* function until the *done* variable goes to 1, which corresponds to the done signal coming from the accelerator. After we exit from the while loop, we can finally perform the validation phase by comparing the output of the accelerator with the golden output calculated by the baremeal as explained before.

5.2 SoC Generation

When the bare-metal application is ready we can proceed with the SoC generation. First of all, from the *socs* directory we need to move into our board folder, in our case the *profpga-xc7v2000t* folder, and then run the *make-esp-xconfig* command on a terminal to open the ESP GUI. Now we have to choose the grid dimension in order to accommodate our Depthwise accelerator and then, we need to select the 64-bit Ariane processor because we used a $DMA\_WIDTH = 64$ to build the accelerator. An important thing that we need to highlight is that the the clock frequency of this Ariane CPU is set to 50 MHz in ESP. This is a very low frequency for a modern CPU, but we have to remember that this entire design flow is meant to create a SoC that will be implemented on a FPGA. The low frequency is required because the CPU will be emulated by the FPGA. Now we can choose the Depthwise accelerator tile directly from the GUI. In Fig. 5.9 we can see the final configuration of the GUI to design a single-core Depthwise accelerator SoC.


General SoC configuration:
virtex7
ETH FPnew
No JTAG
Eth (192.168.1.9)
No SGMII
SVGA+FB
With synchronizers

CPU Architecture:
Core: ariane

Shared Local Memory:
KB per tile: 256

Data transfers:
Bigphysical area
[✓ Scatter/Gather]

Cache Configuration:
Use Caches: ☐
Implementation: ESP RTL
L2 SETS: 512
L2 WAYS: 4
LLC SETS (per mem tile): 1024
LLC WAYS: 16
ACC L2 SETS: 512
ACC L2 WAYS: 4

NoC configuration
Rows: 2 Cols: 2
Config
☐ Monitor DDR bandwidth
☐ Monitor memory access
☐ Monitor injection rate
☐ Monitor router ports
☐ Monitor accelerator status
☐ Monitor L2 Hit/Miss
☐ Monitor LLC Hit/Miss
☐ Monitor DVFS

Num CPUs: 1
Num memory controllers: 1
Num local memory tiles using on
Num local memory tiles using off
Num I/O tiles: 1
Num accelerators: 1

Num CLK regions: 1

Num CLKBUF: 0

VF points: 4

NoC Tile Configuration

(0,0): mem — ☐ Has cache ☐ Has DDR — Clk Reg: 0 ☐ Has PLL ☐ CLK BUF
(0,1): IO — ☐ Has cache ☐ Has DDR — Clk Reg: 0 ☐ Has PLL ☐ CLK BUF
(1,0): cpu — ☐ Has cache ☐ Has DDR — Clk Reg: 0 ☐ Has PLL ☐ CLK BUF
(1,1): DEPTHWISE_CXX_CATAPULT Impl.: asic_fx32_dma6 — ☐ Has cache ☐ Has DDR — Clk Reg: 0 ☐ Has PLL ☐ CLK BUF

Figure 5.9: Depthwise accelerator GUI configuration

The last thing that we need to do is to press the *Generate SoC Config* push-button to apply the changes and build the SoC.

5.3 SoC Simulation

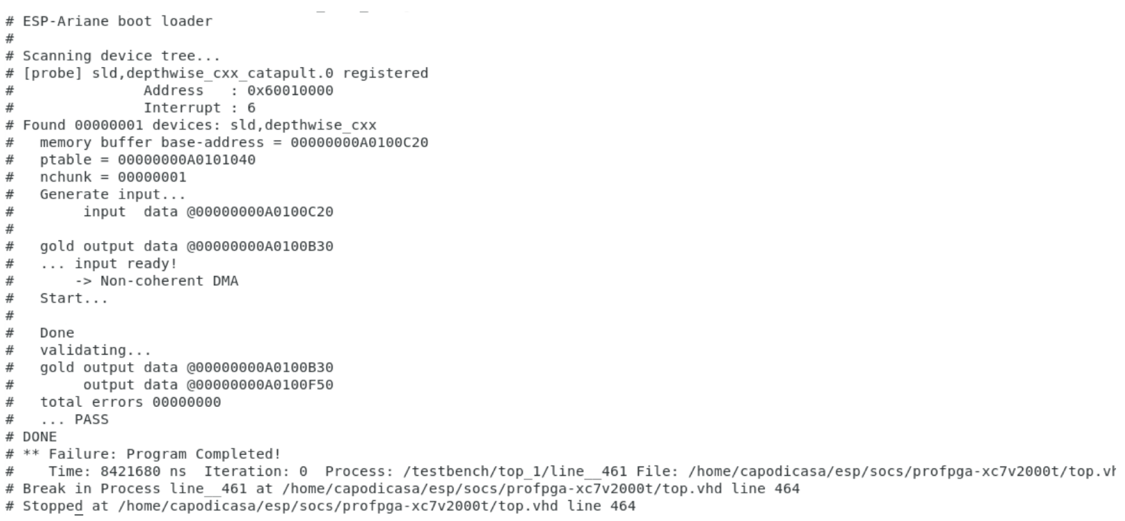
In order to test the behavior of our SoC, we can run a full-system RTL simulation in Questasim running the following dedicated ESP make scripts on the terminal. First of all we need to compile the baremetal application with the following command:

$$\textit{make depthwise\_cxx-baremetal} \tag{5.1}$$

Then we have to run the actual simulation with the *make qsim* command specifying the test program with:

$$\textit{TEST\_PROGRAM=./soft-build/ariane/baremetal/depthwise\_cxx\_catapult.exe} \tag{5.2}$$

After the simulation process we will see on the terminal an outcome like the one in Fig. 5.10.



```
# ESP-Ariane boot loader
#
# Scanning device tree...
# [probe] sld,depthwise_cxx_catapult.0 registered
#              Address   : 0x60010000
#              Interrupt : 6
# Found 00000001 devices: sld,depthwise_cxx
#   memory buffer base-address = 00000000A0100C20
#   ptable = 00000000A0101040
#   nchunk = 00000001
#   Generate input...
#        input  data @00000000A0100C20
#
#   gold output data @00000000A0100B30
#   ... input ready!
#       -> Non-coherent DMA
#   Start...
#
#   Done
#   validating...
#   gold output data @00000000A0100B30
#        output data @00000000A0100F50
#   total errors 00000000
#   ... PASS
# DONE
# ** Failure: Program Completed!
#    Time: 8421680 ns  Iteration: 0  Process: /testbench/top_1/line__461 File: /home/capodicasa/esp/socs/profpga-xc7v2000t/top.vh
# Break in Process line__461 at /home/capodicasa/esp/socs/profpga-xc7v2000t/top.vhd line 464
# Stopped at /home/capodicasa/esp/socs/profpga-xc7v2000t/top.vhd line 464
```

Figure 5.10: SoC RTL Simulation

If the simulation ends without errors, the outcome will be *PASS* otherwise it will be *FAIL* with the associated number of found errors.

5.4 FPGA Implementation

The last step that we need to do is the implementation of the SoC on FPGA. ESP provides a dedicated integration flow that allows to create a bitstream, program the FPGA and run the bare-metal application on the board with some simple make commands. In Fig. 5.11 we can see our target board: the Profpga-xc7v200t.

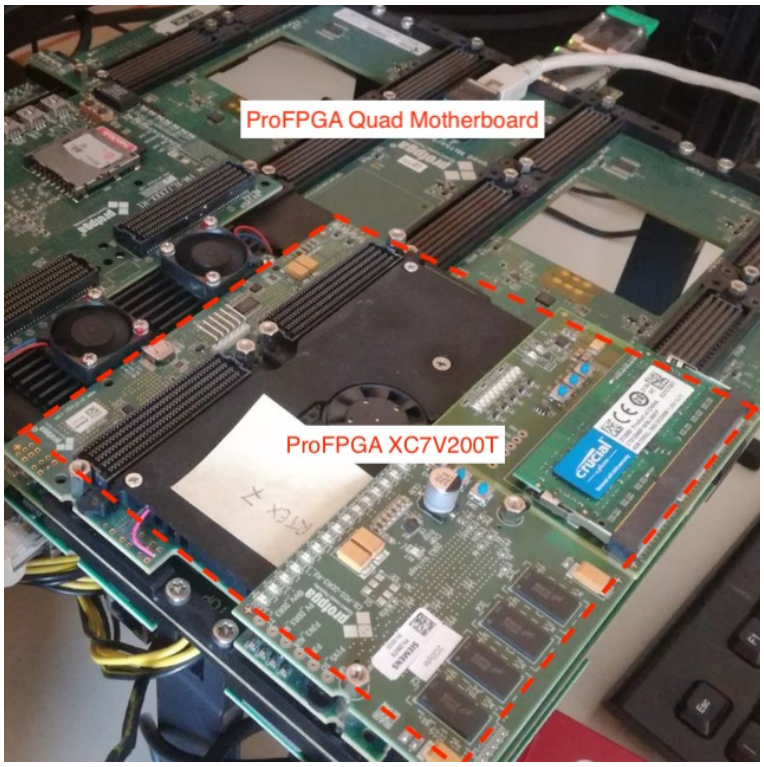
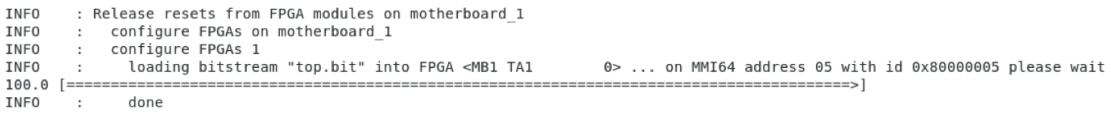


Figure 5.11: ProFPGA XC7V2000T connected to a ProFPGA Quad Motherboard

First we need to generate the bitstream. This can be done running the *make vivado-syn* command. At the end of this quite long process a file in the board folder called *top.bit* is generated: it is the actual bitstream file. Then we can program the FPGA running the *make fpga-program* command. The outcome from the shell will be the one showed in Fig. 5.12.



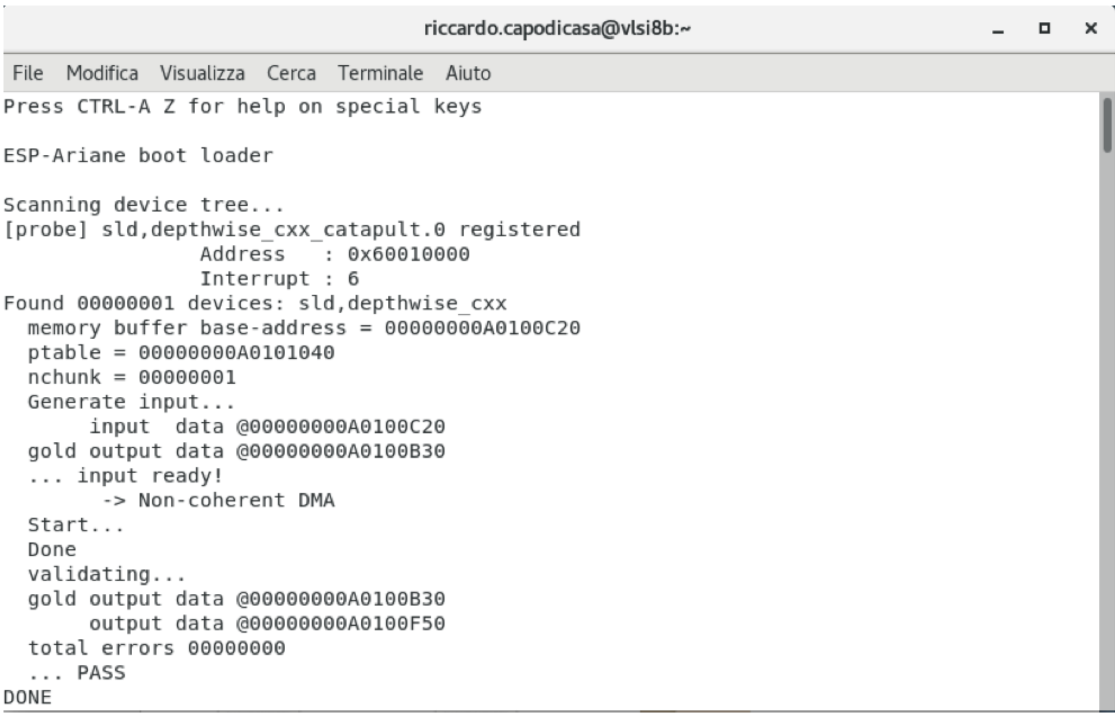
```
INFO    : Release resets from FPGA modules on motherboard_1
INFO    :   configure FPGAs on motherboard_1
INFO    :   configure FPGAs 1
INFO    :     loading bitstream "top.bit" into FPGA <MB1 TA1       0> ... on MMI64 address 05 with id 0x80000005 please wait
100.0 [=======================================================================================>]
INFO    :     done
```

Figure 5.12: FPGA Programming

Finally, we can run the bare-metal application directly on the FPGA. This can be done with the *make fpga-run* command specifying the test program with:

$$TEST\_PROGRAM=./soft\text{-}build/ariane/baremetal/depthwise\_cxx\_catapult.exe \tag{5.3}$$

In order to see the results coming from the FPGA, we need a tool to print on screen the data coming from the UART serial interface. In Fig. 5.13 we can see the outcome obtained during the RTL Simulation from a terminal running the *minicom* program.


riccardo.capodicasa@vlsi8b:~
File Modifica Visualizza Cerca Terminale Aiuto

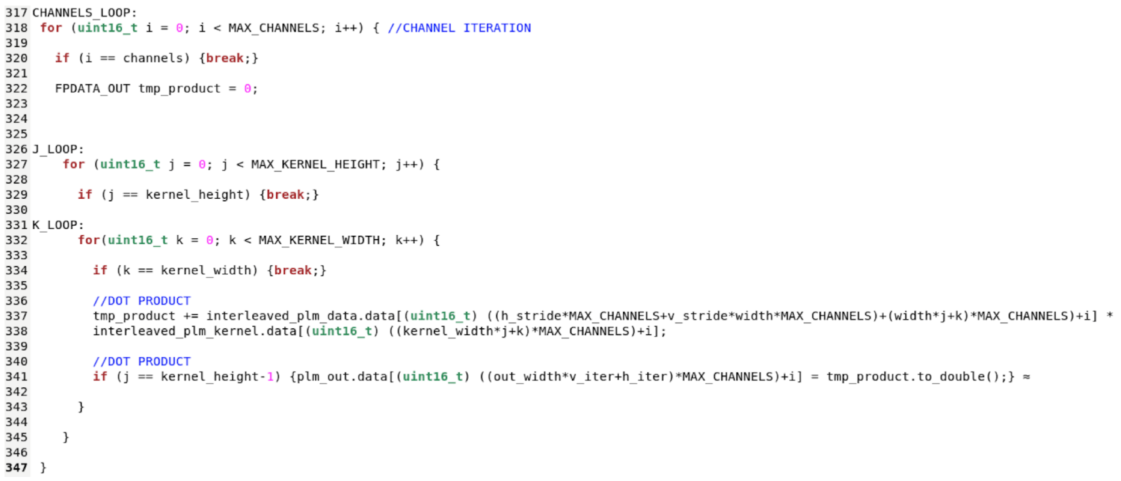
```
Press CTRL-A Z for help on special keys

ESP-Ariane boot loader

Scanning device tree...
[probe] sld,depthwise_cxx_catapult.0 registered
                Address   : 0x60010000
                Interrupt : 6
Found 00000001 devices: sld,depthwise_cxx
  memory buffer base-address = 00000000A0100C20
  ptable = 00000000A0101040
  nchunk = 00000001
  Generate input...
       input  data @00000000A0100C20
  gold output data @00000000A0100B30
  ... input ready!
        -> Non-coherent DMA
  Start...
  Done
  validating...
  gold output data @00000000A0100B30
       output data @00000000A0100F50
  total errors 00000000
  ... PASS
DONE
```

Figure 5.13: FPGA results on *minicom*

Chapter 6

Accelerator Design Space Exploration in Catapult HLS

In the final part of this thesis we performed a design space exploration of our Depthwise accelerator using Catapult HLS. The design space exploration consists in adjusting some "design knobs" in order to explore the Latency vs Area space of possible solutions, and find which of them belong to the Pareto optimal curve. The parameters that we want to compare are the Latency vs Area and Latency vs Power Consumption of the Accelerator inside the SoC. Also in this case we want to leverage the high flexibility provided by Catapult HLS and ESP. With Catapult HLS we applied several directives that allow the designer to choose different optimization techniques and parameters configuration, then we used Vivado in order to perform the logic synthesis and obtain the area parameters. We performed an exploration considering the Channels loop unrolled and the Kernel loops pipelined, varying the maximum number of channels (CH) going from 1 to 16, the Unrolling Factor (UF) going from 1 to 16 and the Initiation Interval (II) going from 1 to 3. Then we set the accelerator parameters for the input width and weight size to their maximum values: 16x16xCH and 5x5xCH, respectively. We can see the chosen design points in the first column of Tab. 6.1. The names mentioned in this design space are those of the variables coming from the C++ code of our Depthwise algorithm. In particular, in Fig. 6.1 we can see the Depthwise convolution kernel of our accelerator, where the for loop over the channels is the external one labeled as *CHANNELS_LOOP* and the for loops over the kernel's dimensions (height and width) are called *J_LOOP* and *K_LOOP*, respectively.



```
CHANNELS_LOOP:
 for (uint16_t i = 0; i < MAX_CHANNELS; i++) { //CHANNEL ITERATION

   if (i == channels) {break;}

   FPDATA_OUT tmp_product = 0;



J_LOOP:
    for (uint16_t j = 0; j < MAX_KERNEL_HEIGHT; j++) {

      if (j == kernel_height) {break;}

K_LOOP:
      for(uint16_t k = 0; k < MAX_KERNEL_WIDTH; k++) {

        if (k == kernel_width) {break;}

        //DOT PRODUCT
        tmp_product += interleaved_plm_data.data[(uint16_t) ((h_stride*MAX_CHANNELS+v_stride*width*MAX_CHANNELS)+(width*j+k)*MAX_CHANNELS)+i] *
        interleaved_plm_kernel.data[(uint16_t) ((kernel_width*j+k)*MAX_CHANNELS)+i];

        //DOT PRODUCT
        if (j == kernel_height-1) {plm_out.data[(uint16_t) ((out_width*v_iter+h_iter)*MAX_CHANNELS)+i] = tmp_product.to_double();} ≈

      }

    }

 }
```

Figure 6.1: Depthwise loops

The UF is a value that represents how many times we replicate a loop in our design in order to increase the concurrency of all the operation inside the loop. For example, in Fig. 6.2 we can see the Catapult HLS Scheduler for an accelerator with 16 Channels and UF=16. In this case we have that the *CHANNELS_LOOP* is replicated 16 times and that all these loops are scheduled in the same clock cycle meaning that they are running in parallel. When a loop is pipelined, the II of a loop is the number of clock cycles after which we can start a new iteration of that loop. For example, if we pipeline a loop with II=1, we can start a new iteration after each clock cycle. We started doing the exploration for the entire SoC, but we noticed that the reports generated by Vivado were always the same. In particular we had the number of LUTs equal to 106618 (8.73%), the number of DSPs equal to 41 (1.90%) and a Power consumption of 4.627 W (3.916 W of Dynamic Power and 0.711 W of Static Power). This happened because that the Area and Power contribution of the Depthwise accelerator inside the SoC is too small with respect to all. the other SoC components. This is why we used the reports generated by Vivado for the accelerator and not for the entire SoC to obtain our results. Below are summarized the results obtained from the exploration.

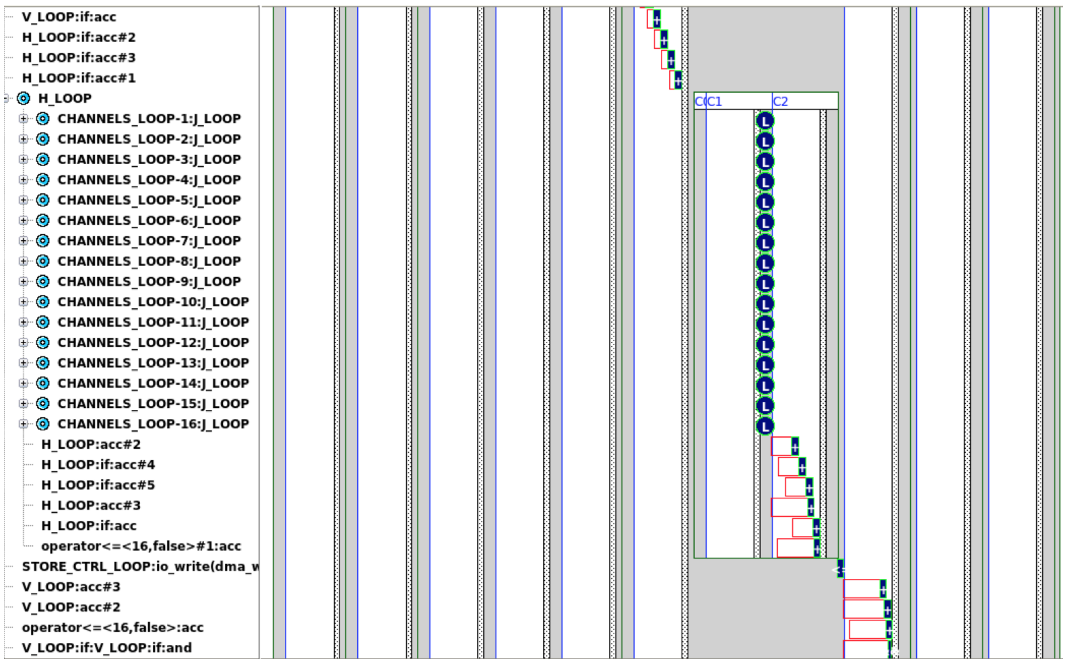


Figure 6.2: Channels Loop in the Catapult HLS Scheduler for a design with 16 Channels and UF=16 design

As we can see from Tab. 6.1, the latency increases with the maximum number of channels and also with the II, but it decreases when we decide to unroll the design, and higher is the UF, lower is the latency. This is valid for all the points except for the solutions with CH=16 and UF=2. This could be due to some sub-optimal internal logic synthesis optimizations made by Vivado. Another thing that we can notice from the table is that the if we increase the UF we will have not only a lower latency, but also an higher resource utilization (as expected), especially in terms of LUTs. This is not always true, in fact if we see the resource utilization differences between the solutions with a different number of channels, we notice that, for example, the single channel solution is almost always the worst in terms of resource utilization. This is because, since it is the simpler solution with the lower latency, Vivado immediately finds a possible implementation that closes timing that does not require further optimizations. We can notice also that the resource utilization in terms of DSPs is almost the same for all the design points with some exception: for example, moving from the solution with CH=4, UF=1, II=1 to the solution with CH=16, UF=1, II=1 we would expect an increase in terms of LUTs, instead as we can see in the table we have a decrease of that number, but almost double the number of DSPs used. It is possible that Vivado optimizes the design looking for a trade-off between the number of LUTs and the number of DSPs.

Design Point	Accelerator Latency (ms)	$DSPs$	$LUTs$
CH=1, UF=1, II=1	0.284	4 (0.1851%)	1590 (0.1302%)
CH=4, UF=1, II=1	1.14	5 (0.2315%)	1132 (0.0927%)
CH=4, UF=2, II=1	1.12	5 (0.2315%)	1165 (0.0954)
CH=4, UF=4, II=1	1.11	5 (0.2315%)	1253 (0.1026)
CH=16, UF=1, II=1	4.47	9 (0.4167%)	1087 (0.0890%)
CH=16, UF=2, II=1	4.52	5 (0.2315%)	1200 (0.0982%)
CH=16, UF=4, II=1	4.41	5 (0.2315%)	1292 (0.1057%)
CH=16, UF=8, II=1	4.38	5 (0.2315%)	1564 (0.1280%)
CH=16, UF=16, II=1	4.37	5 (0.2315%)	2104 (0.1722%)
CH=1, UF=1, II=2	0.446	4 (0.1851%)	1580 (0.1293%)
CH=4, UF=1, II=2	1.79	6 (0.2778%)	1077 (0.0882%)
CH=4, UF=2, II=2	1.77	5 (0.2315%)	1193 (0.0977%)
CH=4, UF=4, II=2	1.76	5 (0.2315%)	1281 (0.1048%)
CH=16, UF=1, II=2	7.06	4 (0.1851%)	1221 (0.0999%)
CH=16, UF=2, II=2	7.11	4 (0.1851%)	1236 (0.1012%)
CH=16, UF=4, II=2	7.0	5 (0.2315%)	1324 (0.1083%)
CH=16, UF=8, II=2	6.97	4 (0.1851%)	1617 (0.1323%)
CH=16, UF=16, II=2	6.96	4 (0.1851%)	2239 (0.1833%)
CH=1, UF=1, II=3	0.59	4 (0.1851%)	1582 (0.1295%)
CH=4, UF=1, II=3	2.38	6 (0.2778%)	1081 (0.0885%)
CH=4, UF=2, II=3	2.37	5 (0.2315%)	1123 (0.0919%)
CH=4, UF=4, II=3	2.36	5 (0.2315%)	1221 (0.0999%)
CH=16, UF=1, II=3	9.44	4 (0.1851%)	1216 (0.0955%)
CH=16, UF=2, II=3	9.49	5 (0.2315%)	1213 (0.0992%)
CH=16, UF=4, II=3	9.39	5 (0.2315%)	1309 (0.1071%)
CH=16, UF=8, II=3	9.35	5 (0.2315%)	1602 (0.1311%)
CH=16, UF=16, II=3	9.34	5 (0.2315%)	2171 (0.1777%)

Table 6.1: Design Points Explored

Finally, we can see how the II impacts on the latency and resource utilization of our designs: if we increase the II, we notice an increment in terms of latency, but also a reduction in terms of resource utilization. So changing this value in the DSE was worth to explore a larger set of design points. However, this is true for all the design points except for the solutions with 16 channels and II=1 that has the lower area with respect to the solutions with II>1. Probably this is because, as we mentioned before, Vivado does not do any additional optimizations to satisfy the timing constraints.

Considering an hypothetical layer characterized by an input tensor of 16x16x512 and a weight tensor of 5x5x512 we can summarize Tab. 6.1 in a plot where we put on the x axis the resource utilization in terms of LUTs and on the y axis the Layer Latency computed as:

$$Layer\_Latency = \frac{Acc\_Latency \cdot Layer\_CH}{Acc\_CH} \tag{6.1}$$

Where: *Acc_Latency* is the accelerator latency reported in Tab. 6.1; *Layer_CH* is the number of channels of the layer, in our case 512; *Acc_CH* is the number of channels of the accelerator, in our case 1, 4 or 16. This is only an example, in general these considerations can be applied also to more complex layers. Finally we draw the Pareto front to connect all the optimal design points.

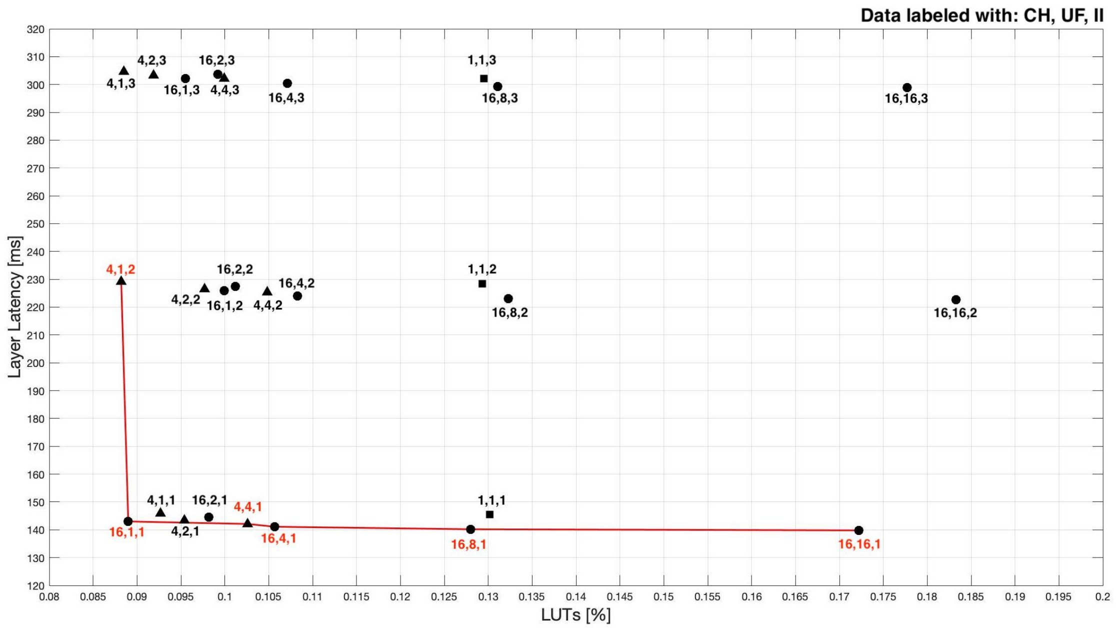


Figure 6.3: LUTs utilization vs Layer Latency design space (Pareto points are connected with a solid line)

From Fig. 6.3 we can see that in order to have a design with a low layer latency (139.8 ms), we have to accept a greater resource utilization (0.1722% of LUTs). On the other hand, if we want a design with a small resource utilization (0.0882% of LUTs), we have to pay with an higher latency (229.1 ms). As we can see the point with CH=16, UF=16 and II=1 has a resource utilization in terms of LUTs of 0.1722% with a Layer Latency of 139.8 ms, but the point with CH=16, UF=1 and II=1 has a resource utilization in terms of LUTs of 0.089% with a Layer Latency of 143 ms. We have that the resource utilization is almost the double with a gain of only 3.8 ms. The SoC designer could use a plot like this to decide which Depthwise accelerator better suits the area budget and/or the latency constraint of the application.

Chapter 7

Conclusions

This thesis work showed how it is possible to create a custom hardware accelerator using ESP and integrate it in a complete SoC in order to test it into an FPGA. In particular, we have analyzed in details the Catapult HLS design flow provided by ESP. For this reason this thesis will contribute to the actual ESP documentation [20]. Then we have performed a Design Space Exploration of our design and we have seen how it is possible to choose between different solutions with different latency and resource usage. This is extremely useful because, thanks to the modularity provided by ESP, the designer can select for her SoC the best accelerator that fits the constraints from the specific application. For example, if we have a tight latency constraints, we can choose the solution that has the lowest latency, but the highest area. Thanks to its simplicity and flexibility ESP is on the good way to become a real standard for heterogeneous SoC integration. In the end, some possible hints and perspectives to expand this thesis work in the next future could be:

- Realize a complete CNN using standard tools like Tensorflow.
- Train the network with a specific data set in order to perform object detection.
- Perform the inference and see the network performances.
- Implement the network in ESP using the hls4ml design flow.
- Substitute the standard convolutional layer with the simple function call of our Depthwise accelerator and measure the latency advantages at network level

Bibliography

[1] LDV Capital. «Five Year Visual Technology Market Analysis: 45 Billion Cameras by 2022 Fuel Business Opportunities». In: (Aug. 2017). URL: `https://www.ldv.co/insights/2017` (cit. on p. 1).

[2] W. Wang; J. Yang; M. Chen; P. Wang. «A Light CNN for End-to-End Car License Plates Detection and Recognition». In: 7 (Nov. 2019), pp. 173875–173883. URL: `https://ieeexplore.ieee.org/document/8915848` (cit. on p. 1).

[3] S. Dabeer; M. Khan; S. Islam. «Cancer diagnosis in histopathological image: CNN based approach». In: 16 (2019). URL: `https://www.sciencedirect.com/science/article/pii/S2352914819301133` (cit. on p. 1).

[4] L. Liu; W. Ouyang; X. Wang; P. Fieguth; J. Chen; X. Liu; M. Pietikäinen. «Deep Learning for Generic Object Detection: A Survey». In: (Sept. 2018) (cit. on pp. 1–3).

[5] V. Sze; Y. Chen; T. Yang; J. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: (Aug. 2017) (cit. on pp. 4, 5, 9–16, 18, 19).

[6] R. Anyoha. «The History of Artificial Intelligence». In: (Aug. 2017). URL: `https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/` (cit. on p. 4).

[7] A. Pandey. «Depth-wise Convolution and Depth-wise Separable Convolution». In: (Sept. 2018). URL: `https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec` (cit. on pp. 20, 21, 23).

[8] A. Howard; M. Zhu; B. Chen; D. Kalenichenko; W. Wang; T. Weyand; M. Andreetto; H. Adam. «MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications». In: (Apr. 2017) (cit. on pp. 20, 22).

[9] P. Mantovani; D. Giri; G. Di Guglielmo; L. Piccolboni; J. Zuckerman; E. G. Cota; M. Petracca; C. Pilato; L. P. Carloni. «Agile SoC Development with Open ESP». In: (Aug. 2022) (cit. on pp. 25–30, 33, 34, 69, 70).

[10] In: (). URL: https://fastmachinelearning.org/hls4ml/# (cit. on p. 25).

[11] «Ariane». In: (). URL: https://github.com/pulp-platform/ariane (cit. on p. 28).

[12] F. Zaruba and L. Benini. «The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology». In: 27 (Nov. 2019), pp. 2629–2640 (cit. on p. 28).

[13] Cobham Gaisler. «LEON3». In: (). URL: www.gaisler.com/index.php/products/processors/leon3 (cit. on p. 28).

[14] D. Giri; P. Mantovani and L. P. Carloni. «Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms». In: (2019), pp. 719–726 (cit. on p. 29).

[15] E. G. Cota; P. Mantovani; G. Di Guglielmo and L. P. Carloni. «An Analysis of Accelerator Coupling in Heterogeneous Architectures». In: (2015), 202:1–202:6 (cit. on pp. 29, 34).

[16] P. Mantovani D. Giri. «How to: design a single-core SoC». In: (Aug. 2021). URL: https://www.esp.cs.columbia.edu/docs/singlecore/singlecore-guide/ (cit. on pp. 30, 31).

[17] H. Mahmood. «The Softmax Function, Simplified». In: (Nov. 2018). URL: https://towardsdatascience.com/softmax-function-simplified-714068bf8156 (cit. on p. 36).

[18] D. Burnette. «Algorithmic C (AC) Datatypes Reference Manual». In: (Aug. 2022). URL: https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf (cit. on p. 46).

[19] D. Giri; P. Mantovani; G. Tombesi; J. Zuckerman. «ESP the open-source SoC platform». In: (). URL: https://www.esp.cs.columbia.edu/docs/specs/esp_address_map.pdf (cit. on p. 71).

[20] D. Giri; P. Mantovani; G. Tombesi; J. Zuckerman. «ESP the open-source SoC platform». In: (). URL: https://www.esp.cs.columbia.edu (cit. on p. 89).