Politecnico di Torino
III Facoltà di Ingegneria

Master's Degree in Electronic Engineering

Master's Degree thesis

FPGA Implementation of a Deep Learning Inference Accelerator for Autonomous vehicles

Supervisor
Ch.mo Prof. Maurizio MARTINA

Candidate name
Giuseppe CESARANO

Candidate ID
235845

Internship tutor Magneti Marelli S.p.A
Ing. Giacomantonio NAPOLETANO

ACADEMIC YEAR 2017/2018

In partnership with

Magneti Marelli S.p.A.

To my grandfather Giuseppe,
who is always guiding and protecting me.

Abstract

The thesis will discuss the implementation of the NVIDIA Deep Learning Accelerator (NVDLA) with FPGA.
The NVDLA is a special purpose accelerator of neural network architectures for deep learning inference, developed by NVIDIA, and whose code has been released by the developers for free. First of all, an overview about deep learning and convolutional neural networks will be given. Then, different categories of accelerators will be introduced, providing examples of applications belonging to each of these categories, and analyzing their performances and applicability in the automotive field (GPUs, manycore architectures, neuromorphic devices and specific purpose accelerators will be considered).
After having considered similarities and differences among the different accelerators, the NVDLA system will be described, highlighting its modularity and configurability. Each single block will be explained and an overview will be given about how the system works. Then, the FPGA chosen by Magneti Marelli for the purpose of this thesis will be introduced: the Zynq Ultrascale+, provided by Xilinx. At this point the integration between the NVDLA and the FPGA will be described.
In the second part of the thesis, the implementation of the NVDLA on the FPGA will be analyzed step by step: first of all, an introduction to the tools used for the thesis will be done, and then both hardware and software implementations will be described. An analysis will be done about the resource utilization, identifying the percentage of resources of the FPGA used by the smallest version of the NVDLA. With this architecture, a very detailed timing analysis will also be performed, taking into account the critical path and the timing closure. Then, the functioning of the system will be verified at different frequencies: the performances will be evaluated, verifying the behavior of the NVDLA considering the different blocks separately and then running a complete neural network architecture, the AlexNet. A comparison with other architectures will be performed.
Finally, different versions of the NVDLA will be analyzed, obtained increasing the size of the different engines, and observing the rise of the FPGA's resource utilization. A complete analysis about the power consumption will be carried out to better understand the performances and to compare the different versions.

Acknowledgments

First of all, I would like to thank my supervisor, professor Maurizio Martina of the Politecnico di Torino, who helped me during all the development of this thesis. He gave me the opportunity to take this thesis in Magneti Marelli and he was always available to answer questions and to solve doubts, and this helped me a lot during these months. Thanks to him, my contribution to the company was very useful and concrete.
I also would like to thank my tutor in Magneti Marelli, ing. Giacomantonio Napoletano, who gave me its support during the development of the thesis. He helped me to solve the issues I had during these months, and he moved me on in the right direction in order to get good results at the end of the implementation of the accelerator.
Of course, I would like to thank the Embedded Systems group, and its responsible Denis Bollea, who gave me the possibility to take part to the Competence Center meetings and activities, allowing me to better understand what does it mean to work in a company like Magneti Marelli, and especially in a team.
A special thank to ing. Andrea Marchese, who helped me during the hardware and software development of the project, devoting me some of his time and sharing with me its knowledge about FPGA that was fundamental to complete the thesis, and to ing. Francesco Ledda, who helped me a lot with the part about the neural network architectures.
Finally, I would like to express my gratitude to my parents and my sister, that helped me a lot during these five years. Even if we were apart, they have never stopped giving me support and encouraging me in finding my way and in realizing my objectives. They have always been close to me, in particular in the difficult times, and I think I will never thank them enough for everything they did for me.
A final special thank to Monica, that gave me the strength to do my best during these years and to overcome each difficulty.

Contents

CHAPTER 1

Introduction

In the last period, most of the car makers are moving toward a new direction: the **autonomous vehicle**. Going on with time, this is getting closer to reality: a lot of car makers are already testing their own autonomous vehicles.
One of the most important aspects related to the autonomous driving technology is about the sensor part, which is fundamental to recognize the different *"obstacles"* that can be found when the vehicle is moving. From this point of view, *artificial intelligence* and *deep learning* are being applied to the automotive field in an increasing way.
The idea of "simulating" the human brain, with both digital (in case of general and specific purpose processors) and analog (in case of neuromorphic architectures) technologies is not a really new idea. What is continuously growing is the type of technology that is used, and the complexity of the algorithm associated to these **neural networks**. For this reason, a lot of processor developers are moving toward this direction. Someone is trying to accelerate neural networks by means of general purpose processors, but nowadays most of the companies are working on new specific purpose architectures, that are only able to work on these networks.
In this context, the **NVIDIA Deep Learning Accelerator** (NVDLA) is one of the novelties introduced by NVIDIA. It is an architecture built for inference, so it can reach very good performances because it is a specific purpose accelerator, that can only perform these kinds of operations. Moreover, the NVDLA is modular, that means that some blocks can be inserted or eliminated, and configurable, so the different engines can be characterized by different sizes, making the architecture adaptable for various scopes.
Another important point is to choose the system that better fits the architecture: the purpose of this thesis is to verify how the **FPGA** (Field Programmable Gate Array) can work in terms of performance and power consumption.
For this reason, the thesis will focus on the FPGA implementation of the NVDLA accelerator, and on its performance analysis.

The RTL description of the accelerator, at least for some versions of the NVDLA, has been released for free by NVIDIA, so the Verilog code is completely available.
Thanks to the availability of the code, a tool, provided by Xilinx, that is the vendor of the Zynq Ultrascale+ FPGA that will be used for this thesis, can be used. The name of this tool is **Vivado**. Its output generates a complete hardware description of the system that can be used by another tool, **Petalinux**, that is responsible of the software implementation. To observe the change in performance, different versions of the NVDLA will be considered, working at different frequencies.

CHAPTER 2

State of art

After introducing definitions and characteristics about the neural networks, this chapter describes some of the architectures available on the market which are able to accelerate neural networks. An overview of the different categories of accelerators will be given, providing examples of applications belonging to each of these categories and analyzing performances and applicability in the automotive field.

2.1 Overview

Neural networks play a critical role in realizing the future of **autonomous driving**, in particular for object detection and classification. As the autonomous driving level increases, there is the need for more complex Neural Networks, able to perform their operations with a very high accuracy and precision, besides the fact that the algorithms must be very fast.
For this reason, using good architectures able to support these neural networks is fundamental, and providing a scouting of them can be useful in order to analyze all possible solutions and to understand which one can be the best.
After a detailed scouting of all possible architectures which are able to satisfy these needs, two big categories have been identified:

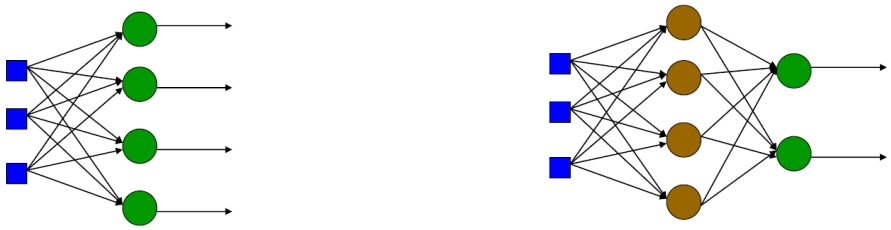
- General purpose architectures, which have not been designed specifically to run a neural network, but they are characterized by a computational power that allows to support a huge number of operations;
- Specific purpose architectures, which can only run a neural network.

Of course the second category has a bigger efficiency with respect to the first one, but the type of operations it can execute is limited. Among these architectures, some subcategories will be considered in the following sections in order to better explain their characteristics.

- For the general purpose accelerators, two categories have been provided: manycore architectures and GPUs;
- For the special purpose accelerators, a focus has been made on the neuromorphic architectures and on the inference accelerators.

2.2 A glance at neural networks development

Neural networks are a branch of machine learning that is inspired to the way the brain is working. The central computational unit is represented by the neurons, which are "simulated" with some computational units. Neurons are organized in **layers**, and the connections between a layer and the following one simulate the **synapses**, that are the elements through which the informations move from a neuron to another one.
A standard architecture can be composed by a number of **input units**, a number of **output units** and, optionally, some **hidden layers**. In figure 2.1 it is possible to see the difference between a single-layer (2.1a) and a multi-layer (2.1b) network: the second one contains the hidden layers.



(a) (b)

Figure 2.1: NN Single and multi-layer architecture: the one on the right is called *3-4-2 network*, due to the number of neurons per each layer

Neurons store the information, while synapses, which are the connections, are characterized by **weights**. In figure 2.1, which represents a **fully connected network**, where each input of the previous layer is connected to all the outputs of the following layer, the information at the output is given by the weighted sum of the inputs:

$$y = \sum_{i=0}^{n-1} w_i x_i$$

Moreover, in order to limit the value of the output of each neuron, an **activation function** is applied:

$$s = \phi(y + b)$$

where b is a bias.
During the learning process, the desired output, which is already known in advance, is specified for a given input. The network, on its own side, calculates the output based on its current weights. These weights, initially, are random. The difference between the desired and the computed output represents the error of the network: thanks to this error, the weights are changed properly of a quantity Δw, which depends on the learning rate.
Of course, this technique can be applied to the single-layer network: multi-layers networks have the hidden layers where the output is unknown. In this case, another rule is considered: the **Backpropagation**. it is an algorithm characterized by different steps.
At the beginning, the error term is computed for all outputs, so at this step the computation is the same of the one performed with the single layer networks; then, from the output layer the error is propagated back to the previous layer and the weights between the two layers are updated, finally, this passage is repeated for each layer until arriving to the inputs of the neural networks
Of course, with this learning method, as the number of layers increases, the complexity of the fully connected exploits at a certain point, so it is impossible to complete the computation. In order to solve this problem, locally connected units have been introduced, increasing the number of layer: in this case we can talk about **Deep Learning**.

2.2.1 Convolutional neural networks

Convolutional neural networks (**CNN**) are an application of deep learning, used in particular for image detection.
Images can be characterized by a very high dimension, so applying a fully connected neural network can be very difficult from the computational point of view. The idea is to "divide" the image in different parts and extract **local features**, that are unified at the end of the process.
CNNs can be characterized mainly by three types of layers: convolution, non-linear (activation) and pooling.

- The convolutional layer is characterized by nothing else that a matrix multiplication between the the input image and a weight matrix called **kernel**. A kernel is a sort of **filter** that "activates" when it sees some specific type of feature at some spatial position in the input. In general, there are different levels of kernels for each convolutional layer [Figure 2.2].

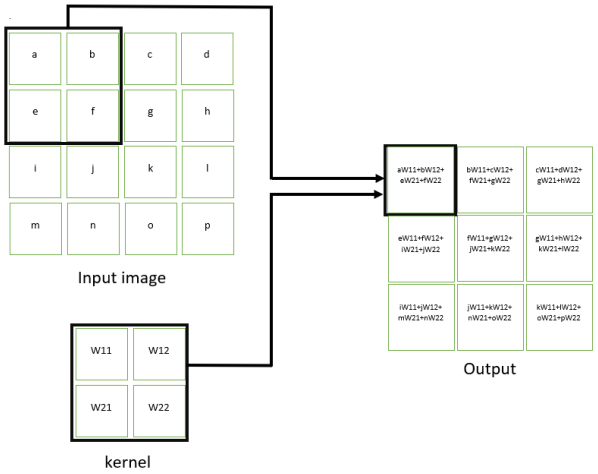


Figure 2.2: Convolutional layer operation

- The non-linear layer increases the non linearity if the architecture. Different functions can be applied, the most important one is the ReLU [Figure 2.3], that has the following equation:

$$\begin{pmatrix} y = 0 & for \;\; x < 0 \\ y = x & for \;\; x \geq 0 \end{pmatrix}$$

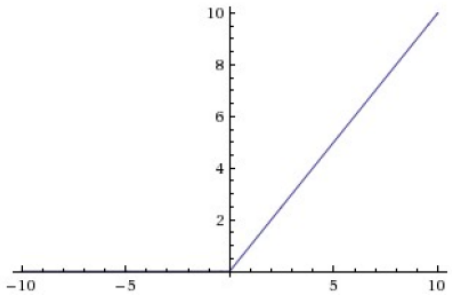


Figure 2.3: ReLU activation function

- The pooling layer is used to reduce the size of the image. The image is divided in windows, and for each of them just one feature is chosen. In the case of the **max pooling** [Figure 2.4] the maximum value of the features is chosen, while in the case of the **average pooling** the average value is computed among all the features of the window.

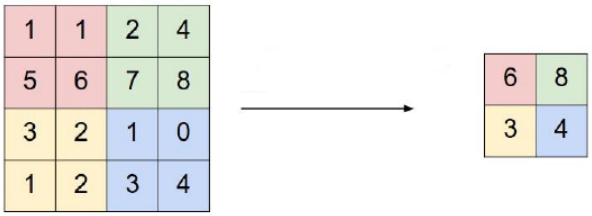


Figure 2.4: Max pooling

In figure 2.5 it is possible to observe the architectures of one of the most important CNNs, the AlexNet.

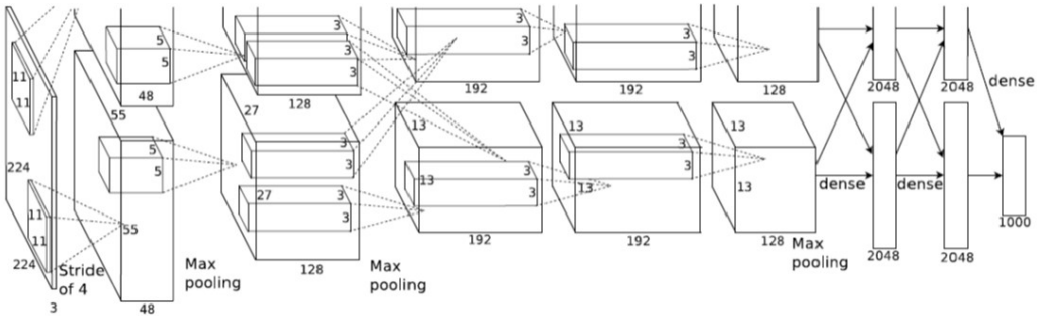


Figure 2.5: AlexNet architecture

The AlexNet architecture was developed by Alex Krizhevsky in 2012 when it won the ImageNet Large Scale Visual Recognition Challenge. It has five convolutional layers and three fully connected layers and an activation function is applied after each layer. The network has 62.3 million parameters and the image size is 224x224.

2.3 Manycore architectures

A Manycore processor architecture is characterized by the presence of a different number of processor cores (from tens up to thousands). They are different from multicore processors, because these last type of processors are characterized by a higher level of parallelism, but they pay in latency and in performance of a single thread.
Communication between the different cores of the processor is obviously the most important issue when dealing with these kinds of architectures. The type of communication which is used in this case is the Network on Chip [Figure 2.6]. This communication protocol is based on some links that physically connect the nodes and actually implement communication, the router which implements the communication protocol (by forwarding packets that are received from some cores to the destination nodes) and Network Interface, which makes the logic connections between the IP cores and the network [1].

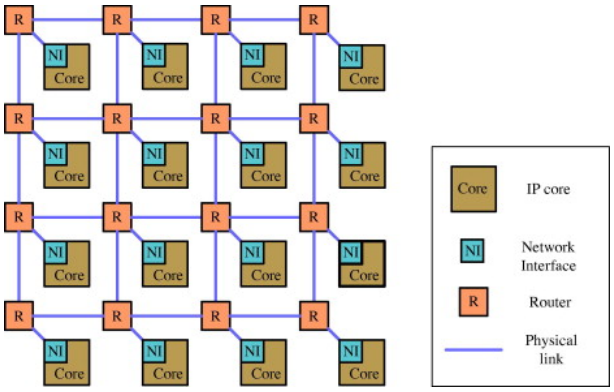


Figure 2.6: Network on chip structure

Manycore processors are general purpose architectures which can be used in a lot of different fields, including automotive. In particular, the high level of parallelism that they provide can give the possibility of accelerating a lot of processes in order to have higher performances. In this context, training a neural network can be computationally intensive and requires a lot of time in case of a limited number of resources. That is the reason why Manycore processors could be used to accelerate neural networks.

2.3.1 Kalray architecture

A good example of manycore processor is represented by the Kalray MPPA-256 family. There are actually three generations of MPPA-256 processors: Andey, Bostan and Coolidge (which is scheduled for the 2018). The Bostan processor, which has been released at the end of 2015, is a 64-bit architecture built with a 28nm technology. It is characterized by sixteen computer clusters plus other two input/output clusters. The I/O cluster are characterized by two quad-cores CPU, which can access directly an external DDR memory. Each computer cluster, instead, has 16 VLIW cores plus one system cores. The total number of cores is 288 [Figure 2.7]

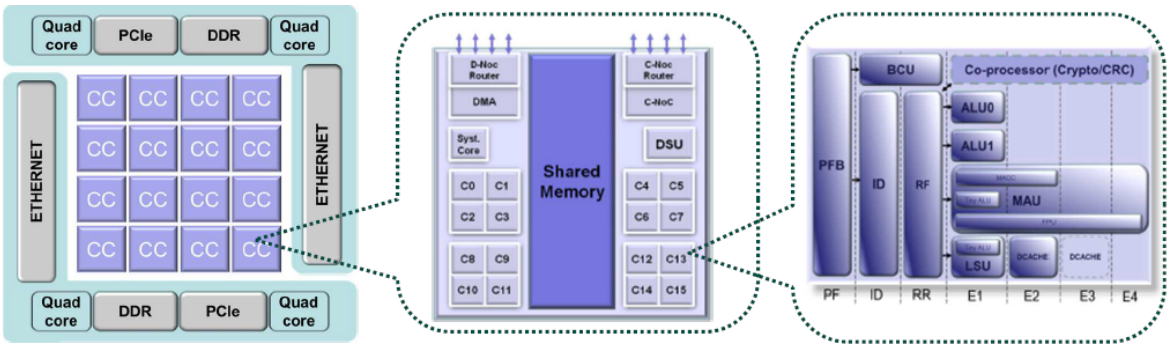


Figure 2.7: Kalray MPPA-256 architecture [25]

The working frequency of the processor is from 400MHz to 600MHz and it can execute about 600 GFLOPS operations at the maximum frequency, with a power consumption up to 25W. About a single cluster, it is characterized by a memory bandwidth of 77GB/s.

Applications in deep learning and neural networks

MPPA architecture is well suited for Deep Learning. In particular, due to the big number of cores, training can be performed and inference can be accelerated once the model has been trained.
Moreover, there is a specific tool, called **KaNN** which is able to take and port any standard deep-learning algorithm (like GoogLeNet, SqueezeNet and others) in Kalrays MPPA processor. Kalray KaNN tool interprets Berkeley Caffe files of trained networks and generates code. The example of the Kalray can be considered in order to understand how the manycore processor architectures can play a very important role in the acceleration of neural networks. As it can be seen in Figure 2.8, a CNN layer can be spread among the different clusters, having a huge decrease in latency

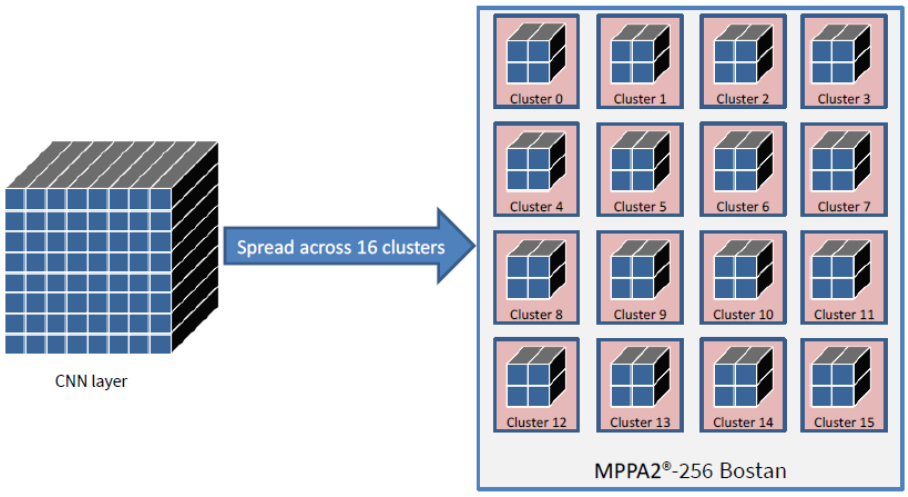


Figure 2.8: Distribution of CNN in the different cores of the Kalray MPPA2-Bostan processor

Applications in automotive

Deep learning and neural network possible utilization is one of the reasons why the MPPA processor can find some applications in the automotive field, in particular for the development of the autonomous vehicle. The MPPA processor supports ASIL B/C and fulfils the ISO 26262 standard.

2.4 GPU

The **Graphic Processing Unit** (GPU) was initially born as a processor used to manage the performance of video and graphics, present in computers but not only there. Nowadays, GPU has reached a power in performances that is so high that is used also for much more difficult computational tasks.
GPUs, as Manycores, are strongly parallel and they are general purpose architectures: the difference is that in this case the cores are smaller but they are much more efficient with respect to the ones present in the manycore architecture. With a manycore processor, each core does its own thing: individual tasks can be run on all of them. In a GPU, instead, there are hundreds or thousands of cores running hundreds or thousands of threads at once. However, there is one important limitation: GPU architecture has to run the same code on all the threads at once.
As said before, nowadays GPU can be used for High Performance Computing and also for **Artificial Intelligence**, because it is able to perform very complex computations.
Due to the huge amount of cores, both training and inference can be performed using GPU. This can be done with the help of a software environment, called **CUDA** (Compute Unified Device Architecture). CUDA is a parallel computing platform and programming model invented by NVIDIA, that provides set of extensions to standard programming languages, like C, that enables implementation of parallel algorithms.
On a GPU a kind of parallelization is deployed, called **topological node parallelization**. This is done using CUDA: in this approach only one copy of the neural network is instantiated. Each thread on the GPU behaves like a single neuron and executes independently. To speed up the implementation, the training weights and input data are stored in a one dimensional array aligned with the host. [2].

2.4.1 NVIDIA VOLTA

A good example that is useful to understand the structure and the application of the GPU for the acceleration of neural network is the new architecture that has been released by NVIDIA. The **VOLTA** GPU is incorporated in the **TESLA V100** accelerator by NVIDIA [Figure 2.9], and it is the first GPU that has been specifically designed to provide a the possibility of executing a huge number of operations in order to train a neural network in a very efficient way, due to the presence of some **Tensor cores** which are able to solve matrix multiplications in a very efficient way.

Figure 2.9: TESLA V100 Module with the VOLTA GV100 GPU [26]

The VOLTA GPU is characterized by six **GPU Processing Clusters**, each of them with seven Texture Processing Clusters. For each of this cluster, there are two **streaming multiprocessors**.
As it is possible to see in Figure 2.10, each multiprocessor has 32 cores executing floating point operations with 64 bits, and 64 cores executing floating point operations with 32 bits, besides 64 INT32 cores. But the real innovation with respect to the previous GPUs, as said before, is the presence of eight Tensor cores for each multiprocessor, so that the total number of cores of the VOLTA GPU is almost 700.

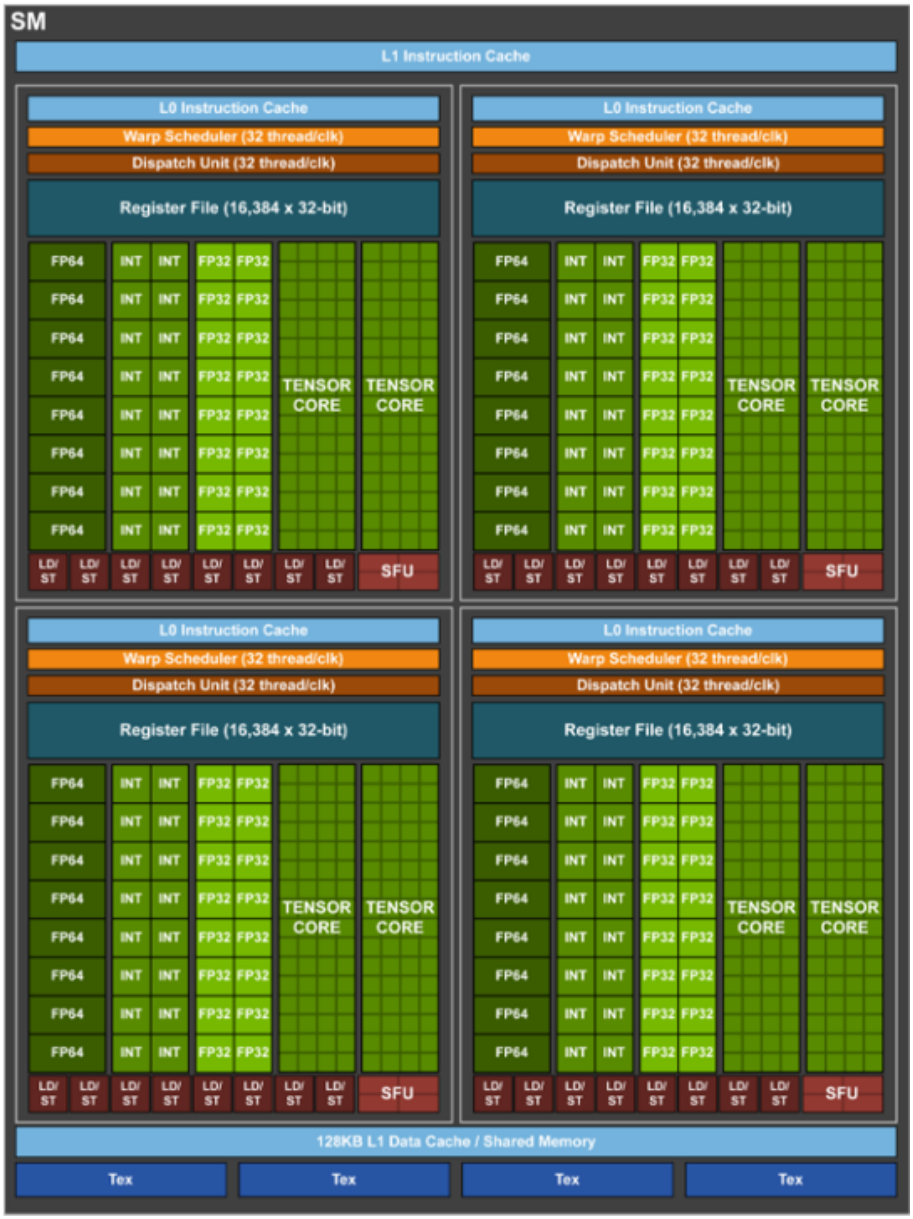


Figure 2.10: VOLTA GPU: streaming multiprocessor [26]

Applications in Neural Networks: Tensor Cores

The tensor cores are able to execute 64 Floating Point **Multiply and Accumulate** operations for each clock cycle, thus reaching a very high performance (about 125 TFLOPs in these tensor cores). Each Tensor Core operates on a 4x4 matrix and performs the following operation:

$$D = A \times B + C$$

The multiplication is a FP16 operation (A and B are matrices containing FP16 values),

while the accumulation can be a FP32 or a FP16 operation. The resulting matrix D has FP32 values [Figure 2.11]. In practice, Tensor Cores are used to perform much larger 2D or higher dimensional matrix operations.

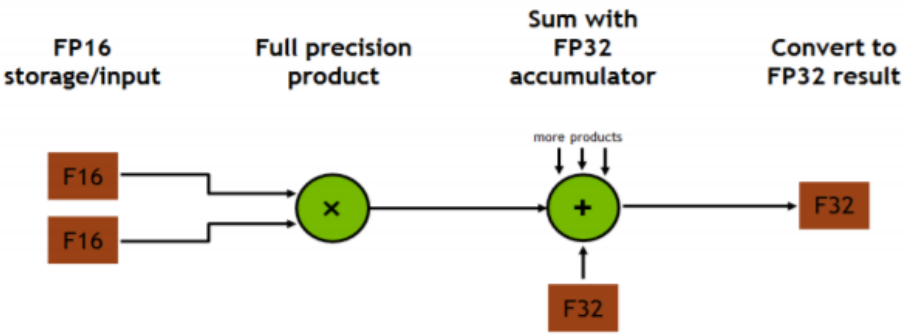


Figure 2.11: Tensor core operations [26]

In Figure 2.12 it is possible to understand the number of outputs which are generated with these cores, which are the results of a 4x4x4 matrix multiply.

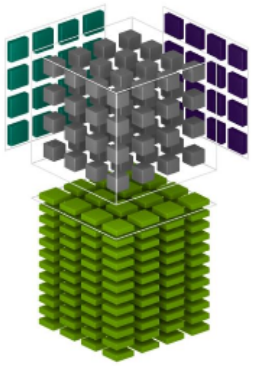


Figure 2.12: Output generation from tensors [26]

The matrix multiplications are the bases for the development of each Convolutional neural network, that is the reason why it was important to underline the possibility that VOLTA has in order to accelerate these operations.

2.5 Neuromorphic architectures

Neuromorphic architectures are structures that are similar to biological brains: they implement artificial neural networks in hardware.
Actually, these architectures are not so recent. The term neuromorphic was coined by Carver Mead in 1990: he referred to *VLSI systems with* ***analog components*** *able to mimic the biological neurons of the human beings* [3]. Nowadays the meaning is quite different, but the

idea of the analog components is still present in the implementation.
The motivations for the developments of these architectures are a lot [5]:

- the need for **low-power consumption** devices;
- the aim to replicate biological networks;
- the possibility to create **real time systems** due to the high parallelism which can be obtained by simulating a huge number (not the exact number, it would be impossible) of the neurons that work together in the human brain;
- the speed of computation which follows the consideration already done for the previous point;
- the possibility to eliminate the latency that the standard von Neumann architecture can have. About this point, one of the problems of the von Neumann architecture is determined by the fact that the memory and the computational part are separated, and for this reason most of the time is spent in moving data from the memory to the computational region and viceversa. With the neuromorphic architecture, since the idea is to **locate the memory really close to the computational part**, this issue could be overcome.

2.5.1 Neuron model

In a human brain, the informations can move from a neuron to another one by means of axons, dendrites and synapses [Figure 2.13]. A neuron accumulates charge through a change in voltage potential across the neurons cell membrane. The voltage potential may reach a particular threshold, after which the neuron fires. In other words, at this point a signal is sent from a neuron to another by means of a spike. The synapses are the connection points between two neurons, and the axons are the lines through which the signal, storing the information, flows.

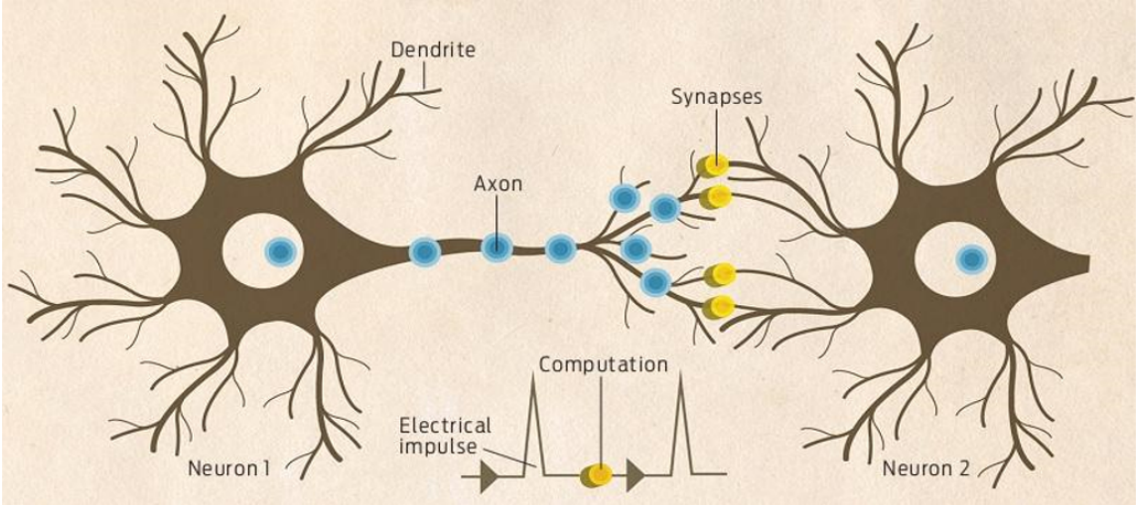


Figure 2.13: Communications between two neurons in a human brain [27]

The concept of accumulating potential and firing is at the base of the structure of the neuron models, which can be more **biologically plausible** or **biologically inspired**. In the first case, the model tries to replicate exactly the biological neuron, for what concerns the structure, so it has neurons, axons, dendrites and synapses. In the second case, neurons models try to model behavior of the neural organization rather than the structure [4].
Among the others, one of the most used in the neuromorphic architectures is the Integrate and Fire model. It is an analog model which can be represented by the classical formula:

$$I(t) = C_m \frac{dV_m(t)}{dt}$$

When a current passes through a neuron, the membrane potential increases up to a certain **spiking threshold**, after which the signal is sent to the closest neuron and the potential is reset to a lower value. This is to show the analog behavior of the neuron model, and that this analog behavior is at the basis of the neuromorphic architectures.

2.5.2 Structure of a general neuromorphic chip

A neuromorphic chip, in general, is characterized by a number of neurons N_c [Figure 2.14], a number of inputs N_{in} [Figure 2.14], and number of synapses per neuron S (the total number of synapses is $N_{in} \times S$).
A **decoder** is the most important part of the chip, because it organizes and distributes the informations to the different neurons. There are two types of decoders:

- **xy-decoder**, in which a single synapse is activated for each input spike, using a column and a row enable line; in this case there is no need to switch off the synapses, because, giving the address made of a row and a column number, the unused ones are simply not accessed;

- **crossbar architecture**, in which each input spike drives a complete synapse column; in this case the architecture must be reconfigurable, and the synapses that have not to be accessed must be switched off [4].

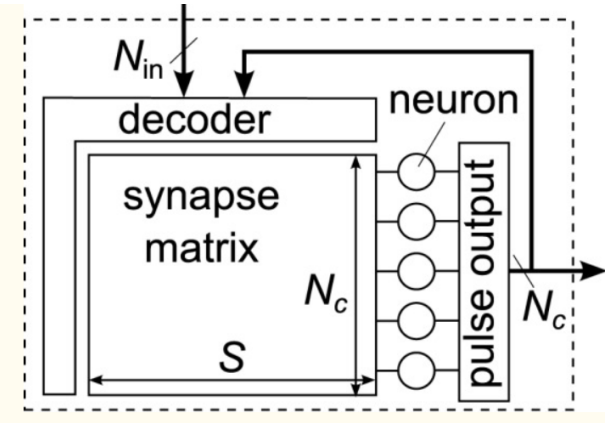


Figure 2.14: Chip of a neuromorphic architecture [4]

2.5.3 Communication in neuromorphic architectures

The communication in neuromorphic architectures must be dealt intra-chip and inter-chip. For the second one, a method is used called **Address Event Representation** (AER) [5]: each neuron has a unique address that is assigned to it, and when a spike must be sent to a specific neuron, its address travels together with the information, that moves through all the adiacent neurons until reaching the target one [Figure 2.15]. The intra-chip communication in general is a simple point-to-point communication.

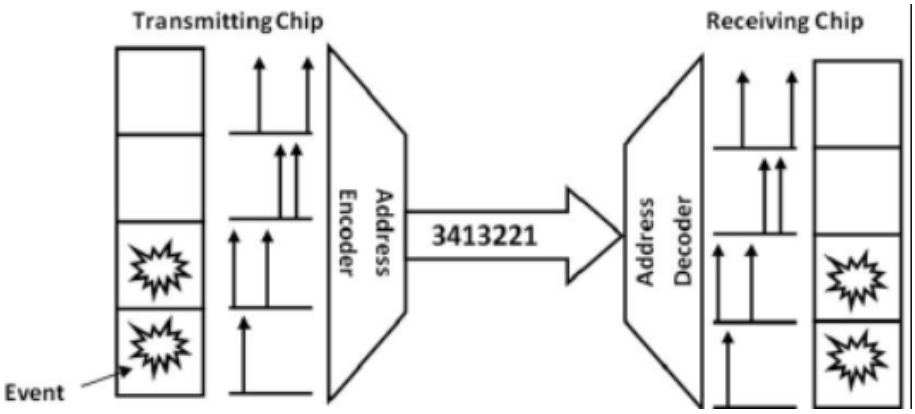


Figure 2.15: Address level representation: a decoder decodes the address sent by the spiking network through an encoder and directs the signal to the right location [6]

2.5.4 SpiNNaker architecture

SpiNNaker is a biologically inspired neuromorphic architecture which was developed by the University of Manchester starting from 2005, even if the first prototype was released in 2009. It was born with the aim of creating a massive parallel spiking neural network architecture which was able to replicate up to a billion of neurons and a trillions of synapses working together in real time. The innovation which was brought by SpiNNaker, which is at the base of the neural networks, is the possibility of transferring a lot of data characterized by a very small dimension (in general, packets are no longer than 72 bits), by means of the Address Event Representation communication. This gives the possibility of operating in a really parallel way.
Actually, a spiNNaker architecture could be interpreted as a sort of manycore processor, because it is characterized by a lot of cores working in parallel. All this cores are general purpose, so, for example, they are not characterized by the synaptic matrix that was described in the previous section. In fact, it is not custom for neuromorphic, but the configuration of each chip includes instructions and data memory in order to minimize access time for frequently used data. SpiNNaker operates in an entirely event-driven fashion to optimize performance and energy consumption. There is no conventional operating system running on the cores. A core is normally in sleep mode (so with low-power consumption). When an interrupt arrives, the core wakes up to perform its own task. At the end of the task, the core returns to sleep. This leads to a low power consumption, that is up to 1W for a single node, while a single board made of a set of nodes consumes more or less from 20W to 50W.

SpiNNaker node

The node is the basic structure of the SpiNNaker architecture.
It has 18 ARM968 processor cores each with 96 kB of local memory, 128 MB of shared memory, a Network on Chip system [7] which performs the communication between the different cores of a single node and with external peripheral [Figure 2.16]. Each node is able to simulate about a thousand of neurons (with about a thousand of synapses for each neuron) [8].
Concerning the cores, one of them has the special role of **Monitor core** and it performs system management tasks, sixteen cores are used to support the application and one is spare. Them, there is a router, called **bespoke multicast router**, from which the signal travels, that is also able to replicate packets where necessary. This is done in order to implement the **multicast function**, that consists in sending the same packets to different destinations.

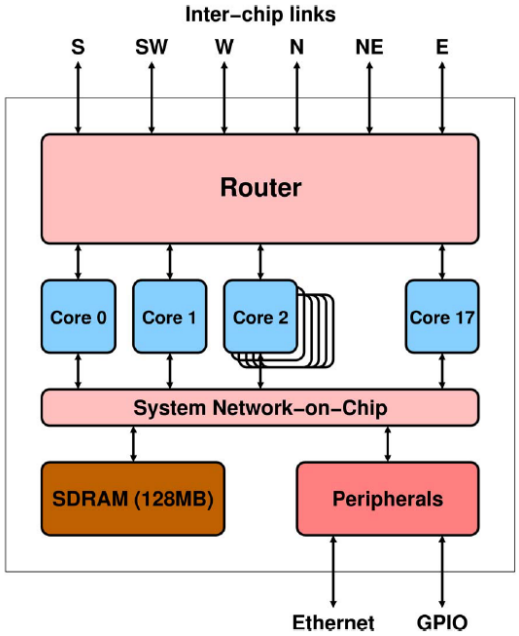


Figure 2.16: Structure of a SpiNNaker node [7]

SpiNNaker system

A SpiNNaker boards is characterized by 48 nodes [Figure 2.17], (so there are 864 ARM processor in total) [7].

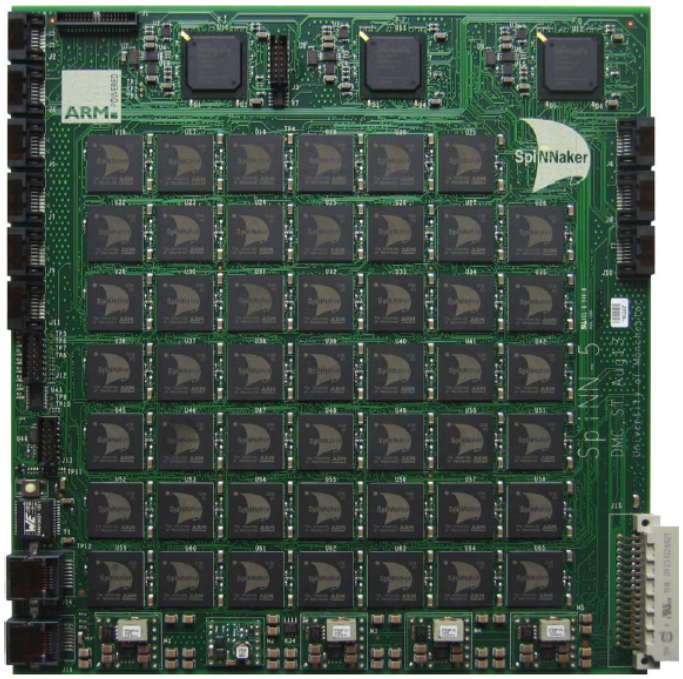


Figure 2.17: SpiNNaker board

There are machines which are characterized by a lot of these boards connected together (up to 1200). They have over a million of ARM processors and they can simulate up to a billion of neurons.

Communication in the SpiNNaker architecture

The basics of the exchange of informations in the SpiNNaker architecture is characterized by the Address Event Representation, with which, thanks to the router, packets are switched from a node to another one. Anyway, inside the architecture, the connections between the different cores inside the nodes and among the different nodes is implemented by means of Network on Chip, which was already present in manycore processors. In particular, the SpiNNaker has two types of NoCs:

- **System Network on Chip**, to handle intra processor communication (data are transmitted through multiple parallel channels);
- **Communication Network on Chip** , to handle inter-processor communication. The bespoke router, which is present also outside of the nodes, connects the neighboring chips (usually in the North, Northeast, East, South, Southwest, and West directions) to form a 2-D triangular toroidal mesh [Figure 2.18] [9].

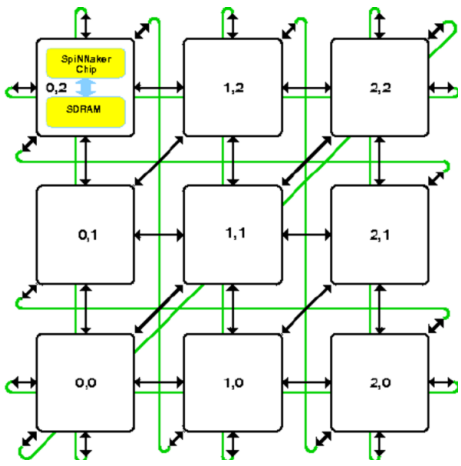
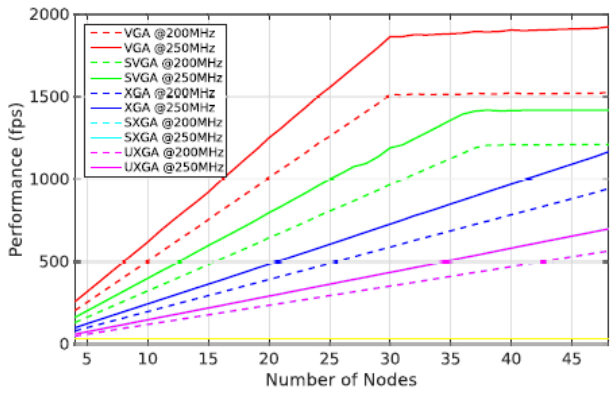


Figure 2.18: SpiNNaker communication among the different nodes

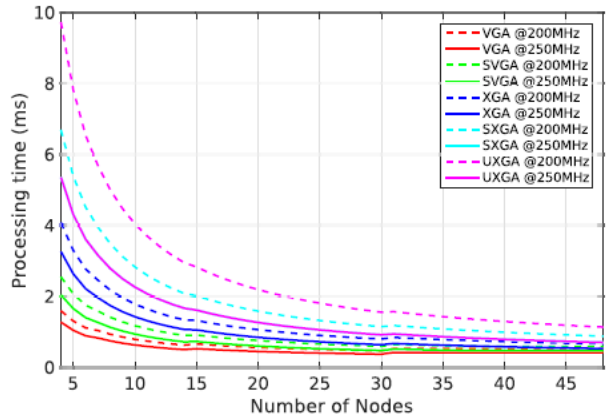
Applications of SpiNNaker: image processing tests

A good application of the SpiNNaker system can be retrieved from [10]. The test was executed on the **SpiNNaker 103 Machine**, that is a 48-nodes board with 864 ARM cores. Two algorithms have been implemented in the context of parallel computing for image processing: **image smoothing** and **edge detection**. Both of them can be performed using convolution.

The processes are executed in parallel and independently from each node. Images used to run the experiment go from VGA to UXGA resolution. The first important thing to notice is the power consumption that increases quite proportionally with respect to the number of resources which are used; in particular, in normal operation at 200MHz, after system boot and without any program running, a SpiNNaker chip consumes only 250mW (the sleep mode that has been described before). Increasing the clock frequency to 250MHz, power consumption reaches about 300 mW. During an intense computation, if all cores are used, the total power consumed by a node can reach 950mW.
About the performances, instead, it has been proved that the speed of computation increases proportionally with the increase of the nodes which are used to perform the computations, up to a certain point. In figure 2.19a it can be noticed that for the VGA image the performance stops increasing after 30 nodes: this happens because an image with VGA resolution of 640 x 480 pixel, can only utilize 480 cores, which can be provided by 30 nodes. The same behavior can be observed more or less in figure 2.19b.



(a) Edge detection performance (in fps) [10]



(b) Gaussian filtering performance (in ms) [10]

2.5.5 Truenorth architecture

Truenorth chip is another kind of neuromorphic architecture which is different from the SpiNNaker, that is the reason why it is important to describe both, in order to have an idea of the different possible implementations.
One of the most important characteristics of Truenorth is the low power consumption (about 65 mW): this is given by the fact that the computational part is completely asynchronous (the communication, instead, is synchronous); moreover it has be designed to be highly scalable, parallel and real-time. It is able to simulate about one million neurons, with 256 millions of synapses [11]. This is obtained considering 4096 different nodes that are tiled in a matrix.

Truenorth node

A single Truenorth node is defined as **neurosynaptic core**. It has the classical structure of a neuromorphic architecture, with a synaptic matrix [Figure 2.20].

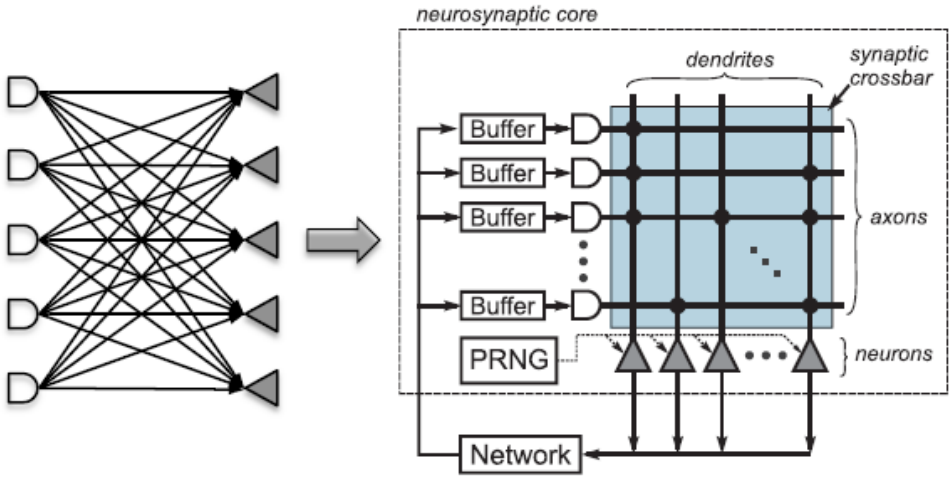


Figure 2.20: Truenorth single node [28]

The exchange of the informations is determined by the following steps [11]:

- the core receives spikes from the network or from the external world, and it sends it to one of the input buffers;
- when a signal arrives, the spike is read from the input buffer and sent to the axons;
- the signal is sent to the whole line; in this case the decoder is of type **crossbar**, so the spike is delivered to the neuron only in the presence of a synapse that is switched on;
- the neuron that receives the spike updates its membrane potential (from the neuron, a value called **leak** will be subtracted at the end of the integration);

- when the membrane potential overcomes the threshold, a spike is generated and sent to another core, and the potential comes back to the lowest level.

This behavior is really analog, and in order to demonstrate it, the formula with which the membrane potential is updated is reported:

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{255} A_i(t) \times w_{i,j} x s_j^{G_j} - \lambda_j$$

A_i is 1 of there is a spike, $w_{i,j}$ is the synaptic weight, s_j is 1 if the synapse is activated and λ is the leak.

Truenorth chip

A complete chip is characterized by an array of 64x64 neurosynaptic cores. They compose the so called neuron block [Figure 2.21]

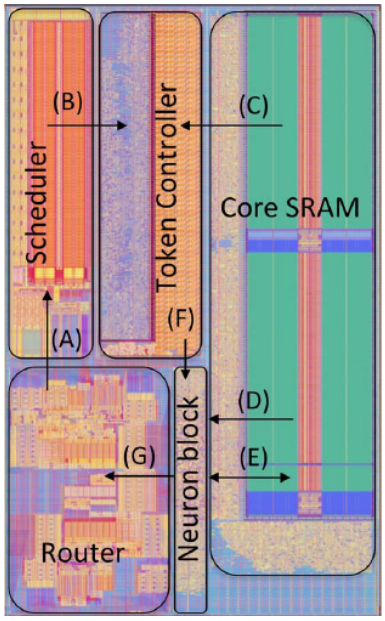


Figure 2.21: Truenorth chip [11]

Looking at figure 2.21, it is possible to observe all the other components of the Truenorth chip.

- There is a router, which communicates with its own core and the four neighboring ones.
- A SRAM stores the spikes as binary values.
- A scheduler takes the data from the SRAM and puts them into a queue waiting for them to be read by the single neurons in the neuron block.
- The controller, has the role of controlling all the neurosynaptic cores.

The architecture could be tiled not only for the single chip, but also for different chips that are connected together [Figure 2.22].

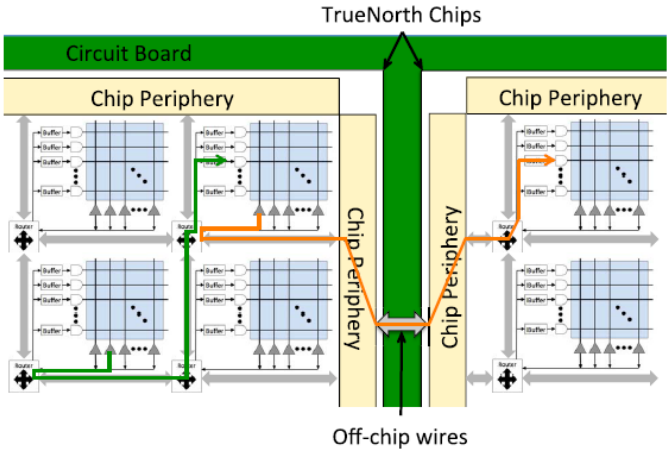


Figure 2.22: Connections between different chips [11]

Applications for Truenorth: object detection tests

An application for the Truenorth chip is presented in [12]. Retrieving images from the camera with a 400-pixel-by-240-pixel aperture, the aim was to identify people, cars, animals etc. The chip consumed 63 mW on a 30-frames-per-second video.
The chip worked in this way:

- the pixels need to be converted into spikes events to interface with the Truenorth;
- two channels are constituted: a high-resolution channel used to identify objects and a low-resolution channel used to locate objects;
- neurons, which were already trained offline to recognize the objects, start to receive spikes;
- bounding boxes are generated.

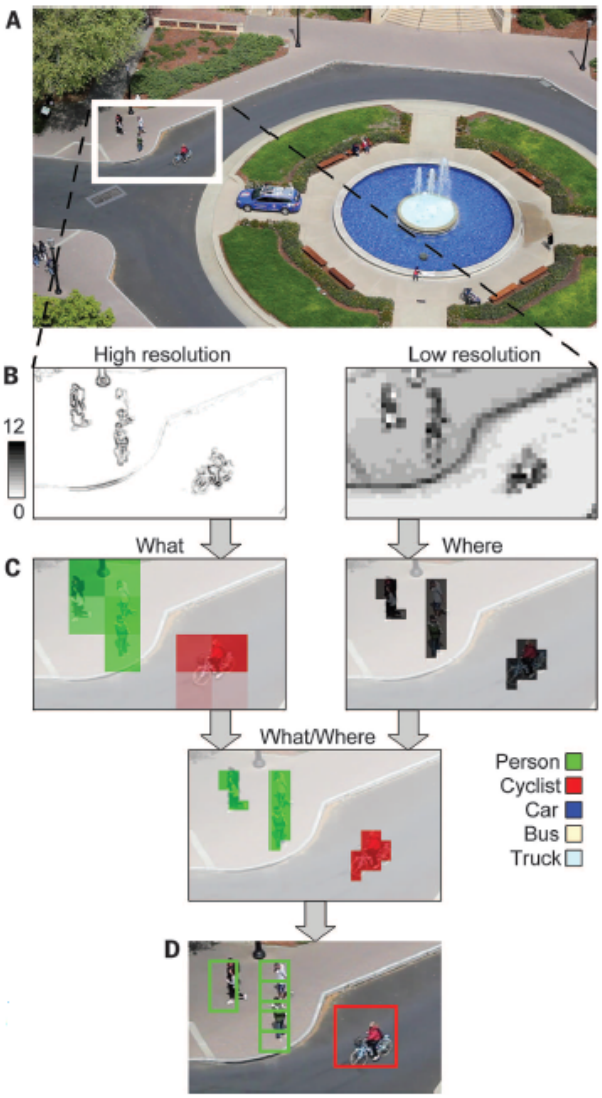


Figure 2.23: Truenorth application [12]

2.6 Neural network accelerators for inference

Traditionally, algorithms based on neural networks could be run on general purpose architectures, like GPUs or Manycore processors, but these architectures in general are characterized by a high power consumption and an oversized resource utilization both for memory and computation. In new applications, where neural networks algorithms have to be exploited, like for example autonomous driving, there is the need for a specific architecture where the low power consumption and the optimization of resources is fundamental.
Obviously, when talking about neural networks, training is the hardest computational part, and in general it is executed on a GPU off-line, in order to have a model that is already trained. Hardware accelerators, instead, can be used for inference, taking the pre-trained

network from the cloud or from an off-chip memory connected to them.
In general, the structure of a specific purpose hardware accelerator is characterized by a buffer memory, where the input images are stored, and another memory to store the weights, then there is the computational part. Since the operations executed during a neural network application are more or less the same, specific processing unit, able to execute convolution, normalization, pooling and all the other layers characteristic of a neural network, are needed. In particular, since the majority of operations in a CNN is represented by a matrix-matrix multiplication, it is fundamental to deal with the massive nested loops in order to increase the throughput [13].

2.6.1 Google TPU

Among the ASICs which are able to perform inference, one of the most important, and maybe the first one to be introduced, is the **TPU**, which stands for **Tensor Processing Unit**, to which Google started to work from 2013 and it was produced in 2015. It is an ASIC used for deep neural networks, built with a 29 nm technology, that works at a frequency of 700 MHz [14].
The main aim of this architecture is to have a performance per Watt that is 15, 30 times smaller with respect to a generic GPU, so the power consumption is very low compared with the general purpose architecture (among 30W).

TPU: general architecture

As it can be seen in Figure 2.24, the architecture of a TPU can be divided in three regions: the control part (highlighted in red), the data and storage part (highlighted in blue) and the computational part (highlighted in yellow).

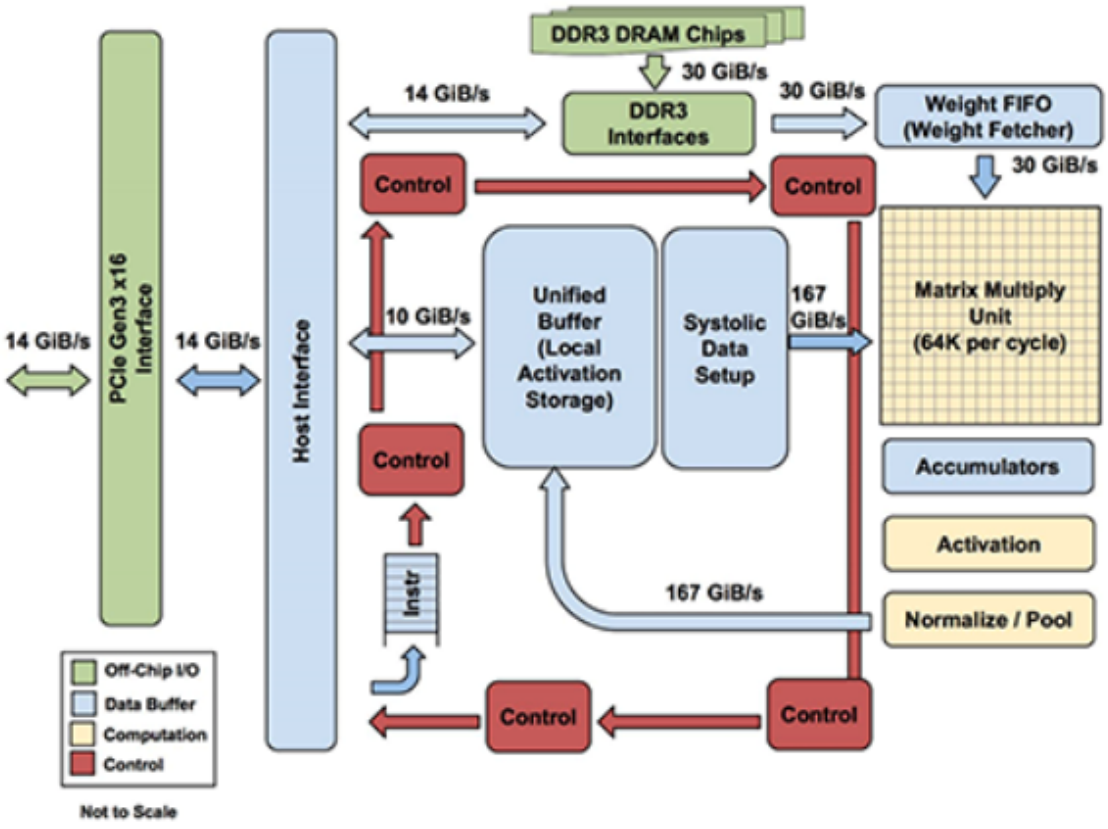


Figure 2.24: TPU architecture [14]

The computational resources include:

- A **Matrix Multiplier Unit** (MXU, that contains 65536 Multiply and Accumulators (256x256). This is the core of the computational part, and the big number of MACs gives the possibility to obtain a very high throughput.
- An **Activation part**, where there is the possibility of applying different activations functions (ReLU, sigmoid etc.).
- An engine where there is the possibility to perform normalization and pooling.

For what concerns the data and storage resource, different units can be identified:

- **Accumulators**, that store the different products coming from the MXU. They are on 32 bits.
- **Weight FIFO**, where the weights are stored. It is on-chip and it reads from an off-chip 8 GB DRAM, called Weight Memory.
- **Unified Buffer**, that is an on-chip 24MB memory that stores the input of the MXU.

One of the most important characteristics of the Google TPU is the usage of the 8-bit integer representations, rather than 16-bit or even 32-bit floating point representation that is used in other architectures. A good efficiency can be also obtained by considering a precision that is not so high, that is why Google adopted this strategy. This is done because the the neural networks must cope noise, and using an 8 bit representation allows to consider only the most important features of an image, that are the ones needed in order to perform a good classification. The 8 bit representation allows the architecture to perform much more operations in parallel.
A high level of parallelism is also reached by considering another technique introduced with the TPU: the **systolic array**. With this technique, matrix multipliers take an input from the Unified buffer and reuse it many times in order to produce all outputs requiring that input. In this case, each value is read only once, but it is used to perform different operations, without storing it back to the buffer.

The parallelism and the low number of bits allow the TPU to have a high performance: in terms of operations, the TPU is able to perform up to 92 TOPS per second.
In Figure 2.25 it is possible to see the performances of the TPU compared with other architectures: the TPU has a long slanted line; this means that performance is limited by memory bandwidth rather than by peak compute. Each NN (represented by stars) runs with a different performance

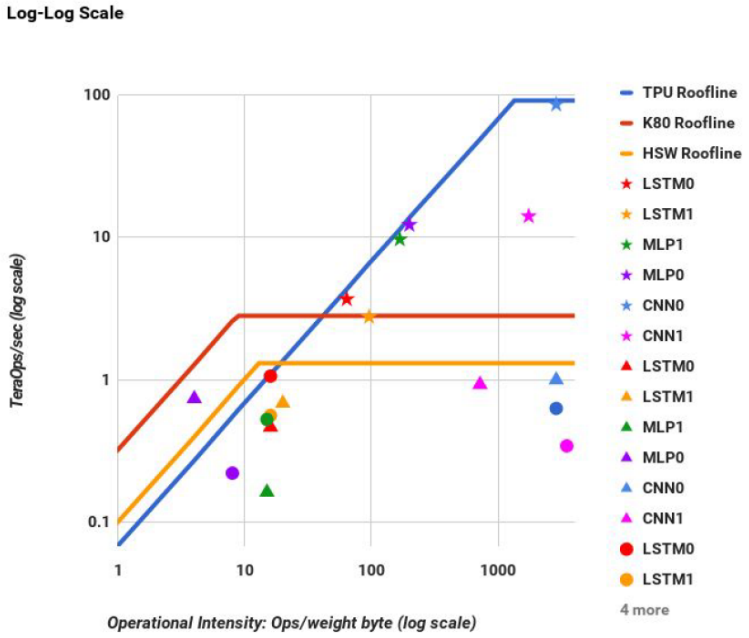


Figure 2.25: TPU performances [14]

The basic idea of the TPU is to make a minimal and deterministic design: the control

part of the complete architecture is just the 2% of the total silicon [Figure 2.26]

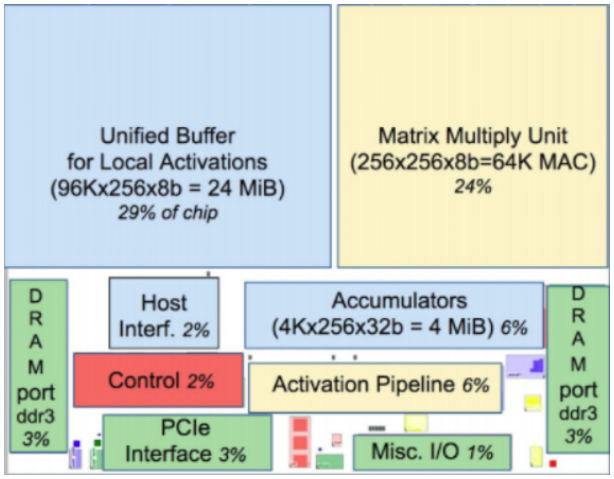


Figure 2.26: Floorplan of the TPU die [15]

Cloud TPU

The Cloud TPU is a second version of the TPU able to perform training and inference. It has a higher performance than the first version of the TPU, because it is able to execute 180 teraflops. It can be programmed with TensorFlow.

2.6.2 EIE: Efficient Inference Interface

One of the most critical problem there is in neural networks is power consumption, that in general is very high in order to guarantee high performances. Thinking to the GPU, the performances are very high because it is possible to perform a huge number of operations in parallel, but the problem is that the power consumption can reach 300W.
A new idea has been proposed by the Stanford University together with NVIDIA [16]: this idea consists in performing a network compression via **pruning** and **weight sharing** that could make possible to fit big networks (such as AlexNet and GoogleNet, that in general are in general very huge) in an on-chip SRAM.
First of all, the representation of the weight matrix changes: for each column of a matrix W, a vector v is stored containing the non-zero weights, then a second one is created encoding the number of zeros between two non-zero elements in the weight matrix.
For example:

$$W_i = [3, 0, 0, 0, 4, 0, 0, 2]$$

can be encoded as:

$$v = [3, 4, 2] \text{ and } z = [0, 3, 2]$$

The accelerator is composed by different **Processing Elements**, operating in parallel. Every processing element stores a partition of network in SRAM and performs the computations associated with that part taking advantage of dynamic input vector sparsity, obtained with pruning. Weight sharing, instead, consists in the modification of the activation function:

$$b_i = ReLU(\sum_{j=0}^{n-1} W_{i,j} a_j)$$

The weight $W_{i,j}$ is substituted with a shared table of 16 weights with an index. This technique is called **Deep Compression**.
In order to parallelize the process, as said before, four processing elements are used. Their usage is explained in the example of figure 2.27.

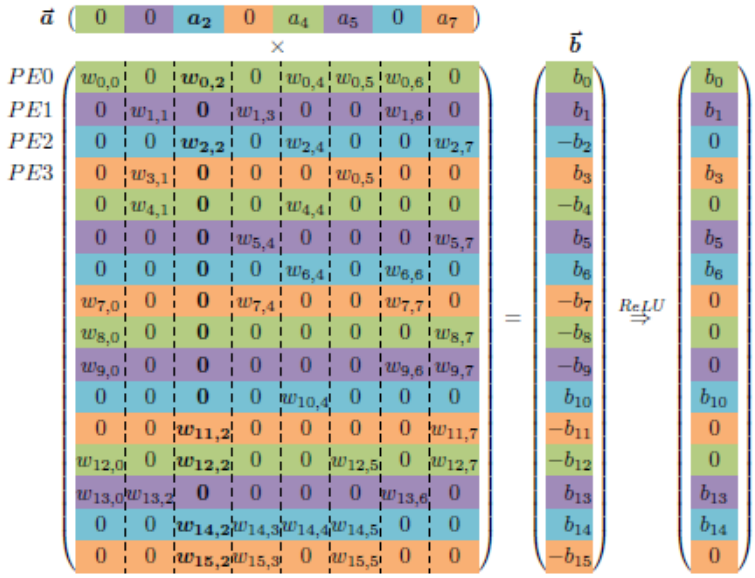


Figure 2.27: EIE representation example [16]

In this example, a vector $\vec{a}$ is multiplied by a weight matrix W in order to obtain a vector $\vec{b}$. Each color represents one **Processing Element** to which the elements of the vectors and of the weigth matrix are assigned. Let us consider the element a_4. This element brings the information related to the fact that it belongs to the fifth column: this information is sent to all the Processing Elements that only perform the multiplication when there is a non-zero element. For example, PE1 in the fifth column has all zeros, so no operation is performed with a_2. At the end, all the results are summed in a row accumulator in order to obtain the corresponding b element.
In order to avoid to perform all the multiplications by 0, a Leading Non Zero detection node is present before each PE.

Performances and energy efficiency

From [16], it can be retrieved that the performance provided by the EIE architecture is more than 10% higher than a GPU. This is obtained because the number of operations to be performed are much lower (about 97% less). Also from the point of view of the energy efficiency, the results are exponentially lower than the ones obtained with a GPU or in general with a general purpose processor.

2.6.3 NVIDIA deep learning accelerator

The **Deep Learning Accelerator** proposed by NVIDIA (called **NVDLA**) is an architecture that is able to perform inference acceleration. The architecture is completely open, in the sense that the Verilog code describing all its components has been released for free by NVIDIA [17].

The most important characteristics of the NVDLA are scalability and modularity. The dimension of all its components can be modified and it is possible to exclude some of them if not needed. The high flexibility of this architecture, the availability of the code and the possibility to explore an accelerator for inference that is completely new, raised the possibility to study it and to work with it in order to understand the performances of the NVDLA and how it could be applied in the possible future in the autonomous driving field as an accelerator of neural networks.

For this reason, NVDLA will be analyzed in the next chapters in more details.

CHAPTER 3

System integration

With this chapter the development part of the thesis begins. First of all, the neural network accelerator which has been chosen to be prototyped is described: the NVIDIA Deep Learning Accelerator (NVDLA). The architecture and the possible applications will be taken into account, also focusing on the possibility of creating a scalable and modular architecture that the NVDLA can provide. Then, a focus will be done on the evaluation board on which the NVDLA has been implemented.

3.1 NVDLA

The **NVDLA** (NVIDIA Deep Learning Accelerator) has been introduced by NVIDIA at the end of 2017. It is an accelerator of neural network for inference, characterized by a high modularity and scalability.
NVDLA is characterized by a hardware implementation that allows to compute the mathematical operations for Deep Learning inference:

- convolution;
- activation;
- pooling;
- normalization.

Moreover, NVDLA is an architecture that is completely open. NVIDIA released the code with some informations about its usage and applications, leaving the users free to add improvements and modifications when required. That is the reason why NVDLA architecture is continuously "evolving" and some novelties are still being introduced.

3.1.1 NVDLA architecture

In order to exploit the modularity of this accelerator, it is useful to analyze its architecture, that is characterized by different components operating separately one from the other [Figure 3.1].

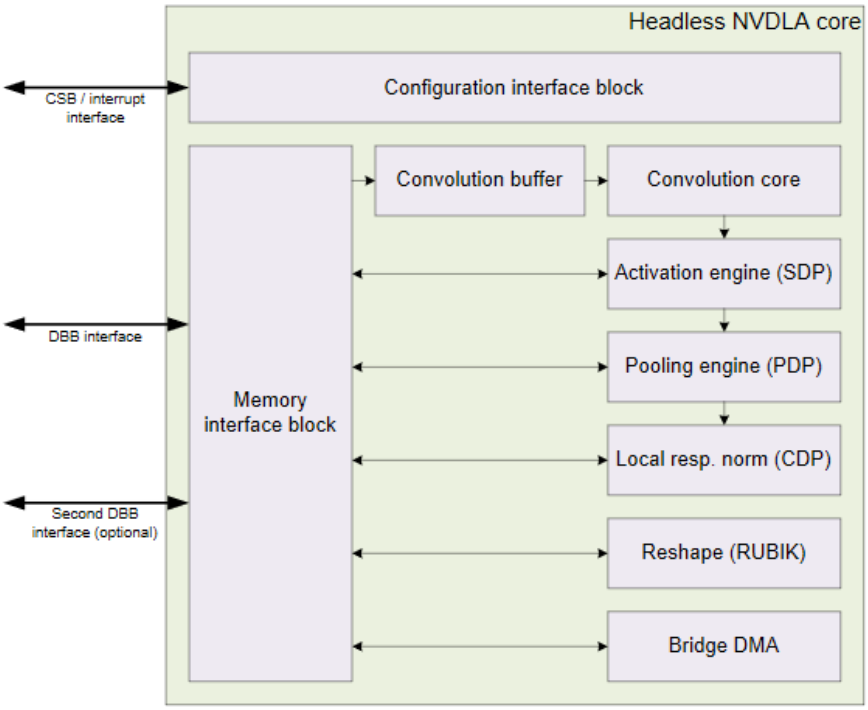


Figure 3.1: NVDLA architecture

- The first component to be considered is the **Convolution core**, that is able to perform high-performance convolutions. It works with two sets of data: the **weights**, that have been already trained off-line, and some **input data**, that can be, for example, the image to be classified. NVDLA supports different kinds of convolutions:
  - Direct and image-input convolutions, that are standard convolutions (with small changes in terms of layers used among them), which are optimized by means of MACs operating in parallel;
  - Winograd and batching convolutions, which are optimizations of the normal con-

volutions obtained by sharing weights and reducing the number of operations.

The convolution engine is supported by a **Convolution Buffer** [Figure 3.1], which stores the weights and the input feature data. It is a normal buffer with two write ports (one for weight and the other for input data) and two read ports.
The complexity of the convolution engine (in terms of number of MACs and number of bits) and the dimension of the convolution buffer can be modified.

- The second component is the **Activation Core** (Single Data Point Operations, SDP). It is used to perform linear and non-linear activation functions. Linear functions are mostly scaling functions, while for non-linear functions there are different supports to ReLU, PReLU, Sigmoid and others.

- The third component is the **Pooling Core**, which is able to perform pooling. Different kinds of pooling are supported, such as max, min, and average pooling.

- The fourth component is the **Normalization Core** (Cross-channel Data Processor, CDP), which is able to apply a type of normalization function called **Local Response Normalization**.

- The fifth component is the **Data Reshape Core** (also called RUBIK), that is able to perform data format transformations, like splitting, slicing, merging, contraction etc.

- Finally, the last component is the **Bridge DMA**. It is a data copy engine to move data between the system DRAM and the dedicated high-performance memory interface, when present.

The NVDLA architecture can operate in two different modes:

- **Independent mode**: all blocks operate independently one with respect to the other, and for each engine there is an access to the memory in order to take the necessary data to perform all operations required;

- **Fused mode**: blocks are assembled as a pipeline; in this case the data are retrieved from memory at the beginning in the first engine and they go back to memory only after the last engine has completed its operations.

3.1.2 NVDLA interfaces

The NVDLA can communicate with the external world by means of the following interfaces [Figure 3.2]:

- a **Data Backbone interface** (DBB), that connects the NVDLA with the external memory. This protocol is very similar to AXI (used by Xilinx with its FPGAs);
- a **high-bandwidth interface** communicating with an external SRAM. This interface is optional;
- a **Configuration Space Bus Interface** (CSB) is a control bus through which a CPU can reach the configuration registers of the NVDLA. CSB is a very simple interface that can be converted to AMBA (supported by the FPGA);
- an **Interrupt interface**, that is a single bit that is asserted when a task is finished or when an error occurs.

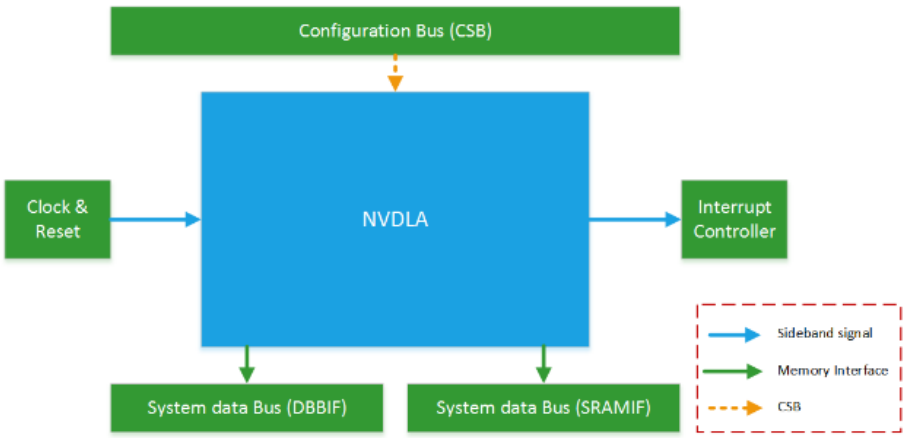


Figure 3.2: NVDLA interfaces

3.1.3 Large NVDLA vs. small NVDLA

The NVDLA scalability property is exploited by observing two opposite implementations from the point of view of the resources: the large NVDLA implementation and the small NVDLA implementation. The difference is expressed in terms of the engines that are used, interfaces with the external system and dimensions of the cores.

Large NVDLA

The large NVDLA is the default version of the accelerator. It is characterized by two memory interfaces: the Data Backbone interface communicates with an external DRAM, while another

interface can be connected to a dedicated high-bandwidth SRAM. The second interface is present in order to satisfy the request for a bigger neural network.
All the engines are present, and the dimension of the convolution buffer is 512MB. Complexity of the different cores will be described in more details in the following chapters.
Finally, the LARGE NVDLA can be considered as a **headed** implementation [Figure 3.3], because it can be supported by a microcontroller which plays the role of executing all the tasks that are related to the high-interrupt-frequency tasks.

Small NVDLA

In the small NVDLA architecture the dimensions of the cores are different, and also some engines are not implemented, like the Rubik engine. Also the Bridge DMA engine is not present, and this is determined by the fact that the second memory interface is absent.
Moreover, in this case we can talk about **headless** implementation [Figure 3.3], because a single processor deals the management of the complete NVDLA hardware.

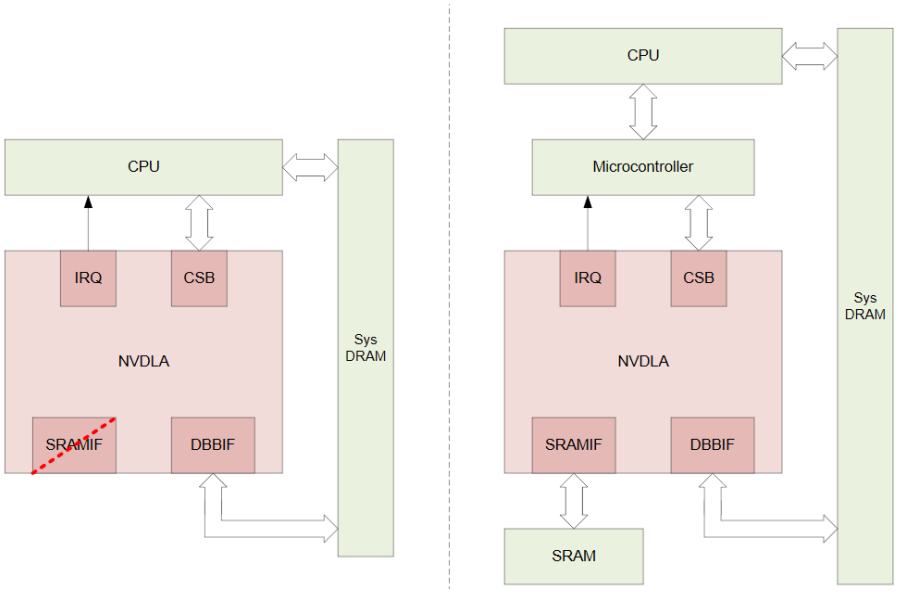


Figure 3.3: Small NVDLA architecture (on the left) and large NVDLA architecture (on the right)

The table 3.1 describes the biggest differences between the two architectures, with an example describing the sizes of the convolution core in the two implementations.

FEATURES	LARGE NVDLA	SMALL NVDLA
Rubik engine	✓	×
Bridge DMA	✓	×
Convolution core size	32×64	8×8
Secondary memory IF	✓	×

Table 3.1

3.1.4 Software design

NVDLA is provided with a software support, that is useful to connect the user with the HW part. In particular, the software related to the NVDLA can be divided in two parts, the **compilation tools** and the **runtime environment**. The compilation tools are characterized by a **parser** and a **compiler**, which start from a Caffe pre-compiled model and generate a network of hardware layers supported by the NVDLA, called **loadable** [Figure 3.4].
The runtime environment instead takes the loadable and run it directly on the NVDLA environment [Figure 3.4]. It can be divided in two different sections:

- the **USER MODE DRIVER** (UMD) provides interfaces to load the loadable produced by the compilation tools and submit it to the lowest level, that is the KMD;
- the **KERNEL MODE DRIVER** (KMD) is characterized by a firmware part and by a set of drivers and firmware that program the NVDLA registers to configure the different engines.

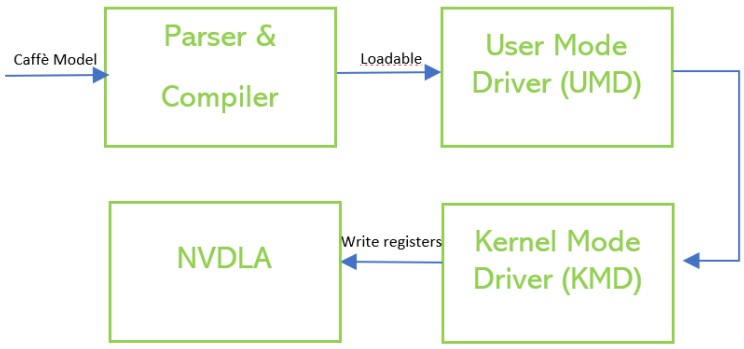


Figure 3.4: Dataflow of the system software

The software design will be explained in more details when talking about the software implementation.

3.2 FPGA and Evaluation Board

FPGA is an optimal solution for prototyping: this is the reason why it is target platform that has been chosen to implement the NVDLA. The FPGA that Magneti Marelli has chosen after a scouting about all platforms with this characteristics, is the **Zynq Ultrascale + MPSoC (Multiprocessor System on Chip)**, by Xilinx.

The Zynq Ultrascale+ family can be characterized by different SoCs, that can change in terms of power and number of resources. The SoC associated to the board chosen as target for the thesis is the **ZU9EG**, belonging to the EG family (quad-core devices).
The Ultrascale family devices are characterized by two different processing systems:

- a Dual/Quad-Core **ARM Cortex-A53 Based Application Processing Unit (APU)**;
- a Dual-core **ARM Cortex-R5 Based Real-Time Processing Unit (RPU)**.

In order just to have an idea of the dimensions of the SoC, a focus can be done on the Programmable Logic: there are almost 600 thousands system logic cells, more that 274 thousands CLB LUTs, more that 2500 DSPs and 32Mb of Block RAM. Figure 3.5 shows the most important characteristics of the FPGA.
The FPGA is mounted on a board, the **ZCU102** [Figure 3.6]. This board has some peripherals that can be useful for the communication between the FPGA and the external world (Ethernet, JTAG, UART, SD reader, USB). Moreover it has a 4GB DDR4 attached to the PS by means of a DDR4 controller.

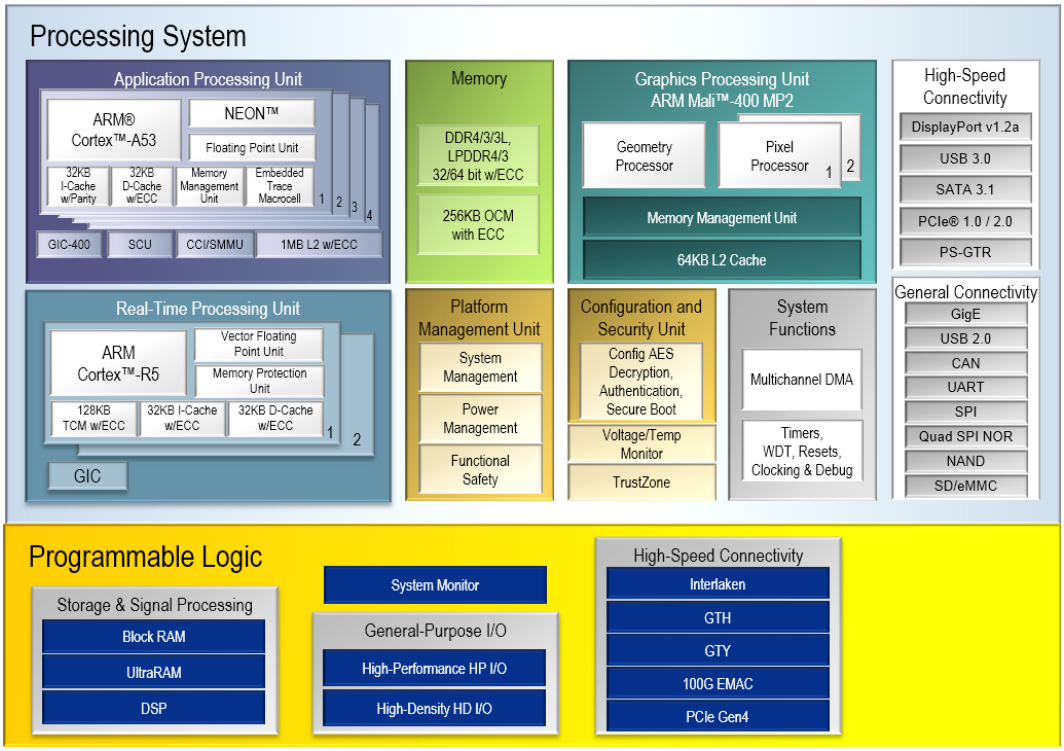


Figure 3.5: Block diagram of the Zynq Ultrascale+ MPSoC FPGA

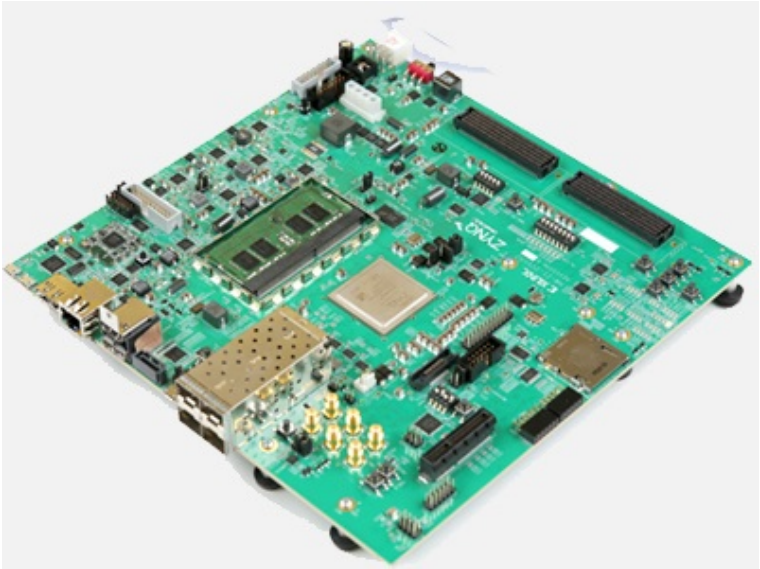


Figure 3.6: ZCU102 board

In this thesis, the FPGA will be programmed by means of the SD reader. The SD card will be provided with an image file containing the Linux OS, that runs on the Cortex A-53 processor, and the executable program together with all data needed to perfom the operations.

3.3 Communication between the NVDLA and the FPGA

As already said, NVDLA uses two standard **AXI bus interfaces** that are used to communicate with memory. In particular, the DBB interface can be connected directly to the DDR4 of the ZCU102 board without passing through the processor [Figure 3.7]. AXI is an interface that is a part of the AMBA, a family of micro-controller buses owned by ARM. **AXI4**, that is the new version of the AXI interface, can be of three different tpyes:

- AXI4, for high-performance memory-mapped communication;
- AXI4-Lite, for low-throughput memory-mapped communication;
- AXI4-Stream, for high-speed streaming data.

AXI4 can support high throughput bursts of up to 256 data transfer cycles with just a single address phase, that is the reason why it is the kind of interface chosen to create the communication between the Zynq and the NVDLA.
The connections are simplified by the fact that the names of all the pins belonging to the DBB port are the same of the ones supported by the Zynq Ultrascale+.

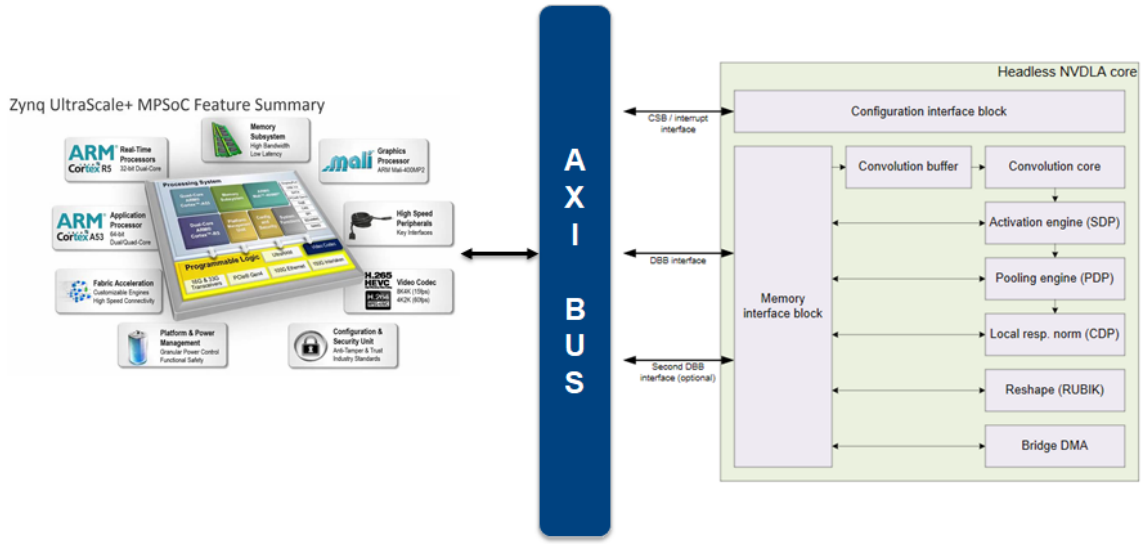


Figure 3.7: Communication between the FPGA and the NVDLA

As it can be noticed in Figure 3.7, also the CSB control interface can communicate with the FPGA by means of the AXI interface: it only needs some **bridges** that change the protocol. In this case, however, the data coming from this interface have to pass through the processing system part, because they are needed by the processor.

CHAPTER 4

Hardware implementation

This chapter describes the hardware implementation of the system it has been introduced in the previous chapter, constituted by the NVDLA and the FPGA. A glance will be given at the tools that have been used to complete this part and then the complete implementation flow will be provided, together with some intermediate results about this part

4.1 Hardware development tools

When performing the development on FPGA, it is possible to start from a high level programming language (C++) that is converted into a hardware description language, like VHDL or Verilog, but it is also possible to begin directly from the Verilog (or VHDL) code. From the description language, a custom block is generated. This is connected to the Zynq FPGA, together with all the components needed to complete the design, then there are synthesis and implementation, with place and route.
Xilinx has developed a set of softwares that are able to perform all these processes. These softwares are all present in an environment called **Vivado Design Suite**. The tool that will be used mostly for the purpose of this thesis is **Vivado Simulator**, that generates an IP (Intellectual property), which corresponds to the block that has the functionalities described by the Verilog (or VHDL) files given as input to the tool. After that, the software is able to perform synthesis, implementation (place, route and optimization) and generate the **bitstream**: this is a binary file containing the complete description of the project, and

it can be exported. The second tool that will be used is **Xilinx SDK**: this tools provides a compilation environment for user applications, but in this thesis it will be used mainly to generate the HDF (Hardware Description) file from the bitstream that has been exported from Vivado Simulator.

4.2 Environment setup

NVDLA provides the Verilog code that is implementing the FULL (Large) version of the NVDLA, that is the default configuration [17]. However, using a Linux environment, there is the possibility of generating a code implementing the SMALL version of the NVDLA, due to the environment setup that NVIDIA provides. The code is generated with a Makefile able to retrieve informations about the characteristics of each core. The specifications for the NVDLA small are the following:

- About the organization of the weights and the input images, all elements are **8 bits** wide;
- About the convolutional block, the MAC atomic size of input channel number and the MAC atomic size of the output kernel number is 8 (that corresponds, respectively, to the "atomic size C" and the "atomic size K") resulting in a number on MACs that is 64 ($K \times C$). The convolution buffer has a depth of 128 KB. Winograd and batch convolutions are not implemented;
- About the other engines, Rubik is disabled, as well as the BDMA;
- The second memory interface is not present.

Details about the NVDLA small specifications are present in the .spec file reported in the appendix (A.1). These specifications can be modified and the code will be generated according to them, that is the reason why the NVDLA is really reconfigurable and it is characterized by a very high modularity. The first *make* command is used to generate a **tree.make** file that is containing all the informations about the NVDLA specifications. Then by performing a simple *tmake*:

```
1 $ ./tools/bin/tmake -build vmod
```

the complete Verilog code, the libraries and all other files required to synthesize the design are automatically generated.

The only problem that has been encountered at this step is about the generation of the SRAMS: they are not generated automatically together with the code (the ones which are generated are related to the FULL architecture, so they are too big), but they are pre-generated, so they are already available in the github project.

4.3 Project setup with Vivado

Once the RTL description of the NVDLA is ready, the Verilog code can be imported in Vivado. The goal is to create the IP, that is the block representing the whole architecture. The RTL generated during the environment setup is actually characterized by two components:

- the NVDLA block;
- an **"APB to CSB"** bridge.

4.3.1 NVDLA block

It is the RTL containing the description of all the functionalities of the NVDLA. Looking at the top entity, that is reported in the appendix (A.2), it is possible to see a big number of inputs and outputs, that can be grouped in the following ports/interfaces:

- the core clock (*dla_core_clk*), the csb configuration clock (*dla_csb_clk*) and a clock used to disable all non-inferred clock gates *global_clk_ovr_on*;
- a port used to disable clock gating when needed (*tmc2slcg_disable_clock_gating*, not used in this project);
- a main functional reset (*dla_reset_rstn*, it is active low) and a reset eventually used during ATPG testing (*direct_reset*);
- a set of ports constituting the **AXI interface** for the memory connection;
- a set of ports constituting the **CSB interface** for the control part;
- a single bit **interrupt** port;
- a set of ports constituting a power control interface (not used in this project);
- a *test_mode* port to enable the test mode (not used in this project).

The complete RTL description, together with the libraries and the RAMs corresponding to the small architecture, is imported in Vivado and it is synthesized, just to see if there are errors in the implementation of the Verilog code. At this point, by means of the command **"Create and Package IP"**, available in Vivado, the IP, that is the functional block containing all

the functionalities of the RTL, is generated. During the generation of the IP, all the ports related to the AXI interface are grouped for simplicity in a unique interface. The IP is visible in Figure 4.1.

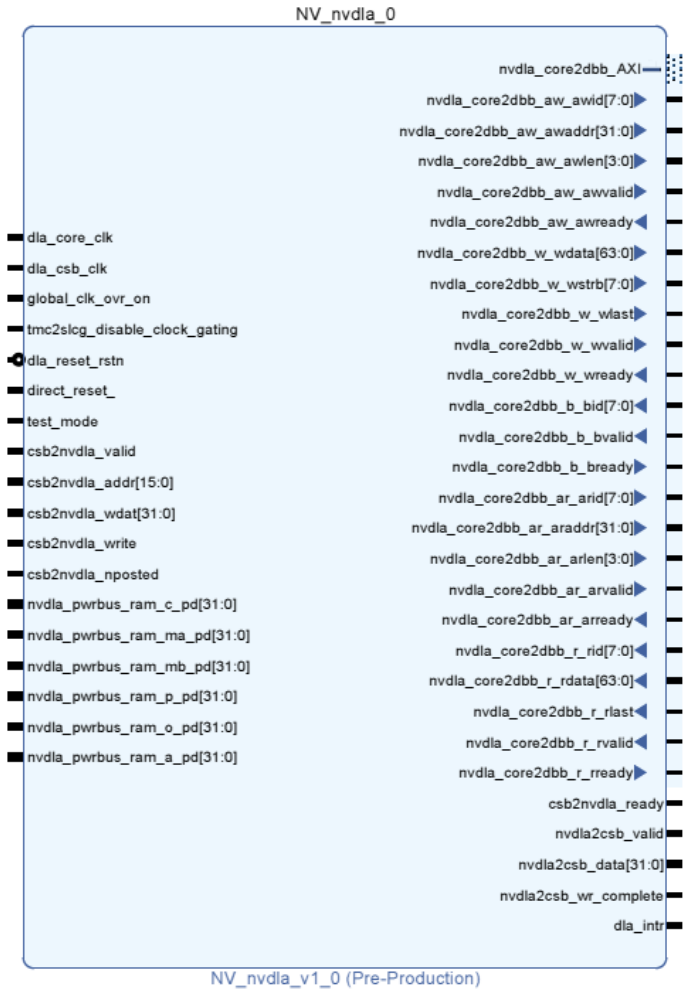


Figure 4.1: NVDLA IP: it can be noticed that the ports related to the DBB interfaces are grouped in a unique interface called **nvdla_core2dbb_AXI**

4.3.2 APB to CSB bridge

The control interface (CSB) provided by the NVDLA is very simple and intentionally low-performance, so that it can be adapted to any kind of processor to which it is connected. However, among the files produced after the environment setup, it is possible to find an RTL description of an element which is able to convert the CSB interface into APB (Advanced Peripheral Bus), that is a family of AMBA bus owned by ARM.

The approach followed in order to create the IP with this functionality is the same of the one used to create the IP of the NVDLA. This time the Verilog code is much simpler (actually it

is a single file, reported in the appendix (A.3)). After packaging it, the IP looks like the one in figure 4.2.

Figure 4.2: APB to CSB IP: apb ports are grouped in a unique APB interface, called **APB_S** (where S stands for *slave*, since it will be a slave port in the top design)

4.4 Creation of the wrapper

After generating these two IPs, and before connecting them to the Zynq, a **wrapper**, containing the two IPs has to be generated. This approach has been followed because of two reasons: the first one is because it is always better two build a hierarchical architecture, so that, when the single IP is required, it can be taken from the design already built and without considering a lot of different RTL descriptions; the second reason is that there is the need to expose some ports and interfaces in the final implementation, that will communicate directly with the FPGA. These interfaces are:

- the clocks;
- the resets;
- the interrupt;
- the memory and control interfaces.

The other ports are hidden inside this design, so they are not visible from the external world. In order to do this, a new project is created in Vivado, with a block diagram representing the two IP connected in the way showed in Figure 4.3.

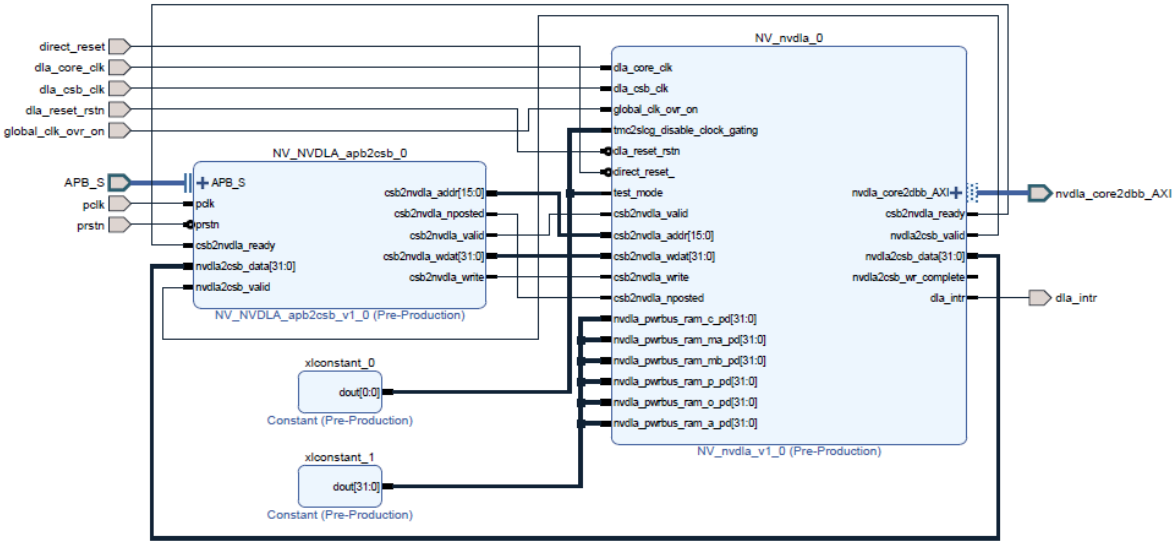


Figure 4.3: Block diagram representing the wrapper obtained connecting the NVDLA and the CSB2APB components

From this picture it can be noticed that all the interfaces and ports to be exposed are characterized by an external port connection. For what concerns the power enable ports, instead, they are connected to a constant block of zeros, because these interfaces are not used in this project, as well as the *test mode* port.
Finally, the connections between the two IPs are very simple because the ports to be connected between each other have the same name.

The following steps are performed to complete the design of the wrapper:

- Validate the design, in order to avoid errors in the connections of the ports in the block diagram;
- Generate the output products, that corresponds to the generation of the different IPs, in an **OUT OF CONTEXT** mode (meaning that the synthesis of the single IPs is performed here, leaving the synthesis of the complete block diagram for the following steps);
- Generate a Verilog wrapper containing an RTL description of the two blocks;
- Synthesize the design, considering the whole wrapper.

Synthesis has been performed just to verify that there is no error in the assignment of the resources.
After that, a new IP containing the wrapper, starting from the RTL description, is generated.

From Figure 4.4 it is possible to observe the ports and the interfaces that will communicate with the FPGA:

- the *dla_core_clk*, the *dla_csb_clk* and the *pclk* ports are the clocks of the wrapper, and the *direct_reset* and *prstn* are its resets;
- the *dla_intr* port represents the interrupt bit;
- *APB_S* is the control interface;
- *nvdla_core2dbb_AXI* is the data interface.

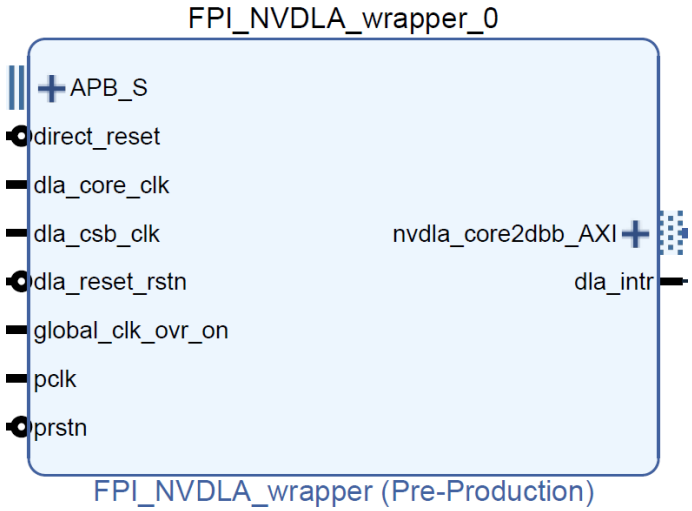


Figure 4.4: NVDLA wrapper IP

4.5 Top design generation

The wrapper IP now has to be connected with the FPGA. The connections are not all direct, and different blocks have been studied and introduced in the design in order to create a correct connection with the Programmable Logic and with Processing System part.

4.5.1 AXI Smart connect

First of all, the Data Backbone Interface has to be connected to the Zynq in **Direct Mode Access** (DMA), in order to accelerate the transfer of data. In fact, with this approach the data move from the DDR4 memory of the ZCU102 board directly to the NVDLA (and viceversa) without passing through the processor. The port of the FPGA through which the DMA can be implemented is the **High Performance** (HP) slave port. This is part of the Programmable Logic of the FPGA, and it supports the AXI protocol.
Even if the DBB interface and the HP slave port both support AXI, DBB has some ports

less with respect to the HP slave port, so a block in the middle has to be put in order to adapt the non corresponding ports: the **AXI Smart Connect** [19].
It is an IP provided by Xilinx that connects one (or more) AXI memory-mapped master devices (in this case the DBB interface) to one (or more) memory-mapped slave devices (HP slave port). AXI SmartConnect is used to automatically configure and adapt master and slave port with minimal user intervention [19].

4.5.2 APB AXI bridge

The APB interface is not directly supported by the Zynq FPGA, that is the reason why another bridge is needed. Xilinx provides an **AXI-APB bridge** that translates AXI4-Lite (slave) into APB (master) [20]. This bridge, eventually, is configurable, so the number of bits for each port can be changed as well as the number of APB interfaces that can be connected to it.

4.5.3 AXI interconnect

The AXI4-Lite interface of the APB-AXI bridge has to be connected to the APU processor, because it represents the control part. In order to do this connection, an AXI interconnect is needed. It is actually very similar to the AXI Smart Connect defined before, with the difference that the AXI interconnect can be used to move from the AXI-4 to AXI4-Lite and viceversa. The AXI interconnect allows the connection between the NVDLA and the Zynq by means of the **High Performance Master** Port in **Full Power Domain** (HPM FDP). This is part of the Programmable Logic of the FPGA and it supports the AXI protocol.

4.5.4 Zynq Ultrascale+ IP

The structure of the Zynq IP is represented in figure 4.5: this IP is highly configurable in order to control the peripherals, to change the clock and to work with the Processing System part and with the Programmable Logic. In figure 4.5 the two connections made with the NVDLA are underlined: it can be noticed that the HPM port is in direct communication with the APU processor (blue path), as the HP port is connected directly to the DDR4 controller (red path), as already explained before.
Vivado simulator helps in creating the connections, in particular for what concerns clocks, resets and interfaces with the same name. However some connections have to be done manually. After building the complete block design, it appears like the one in figure 4.6

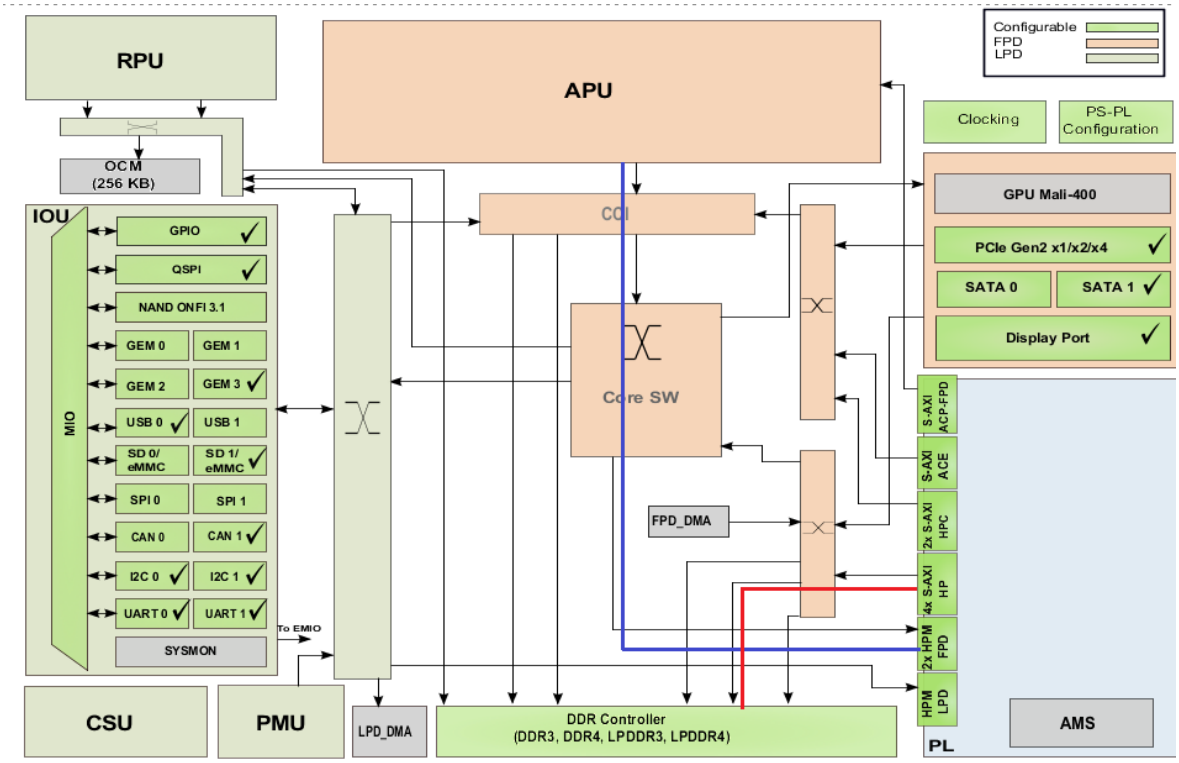


Figure 4.5: Zynq Ultrascale+ block diagram

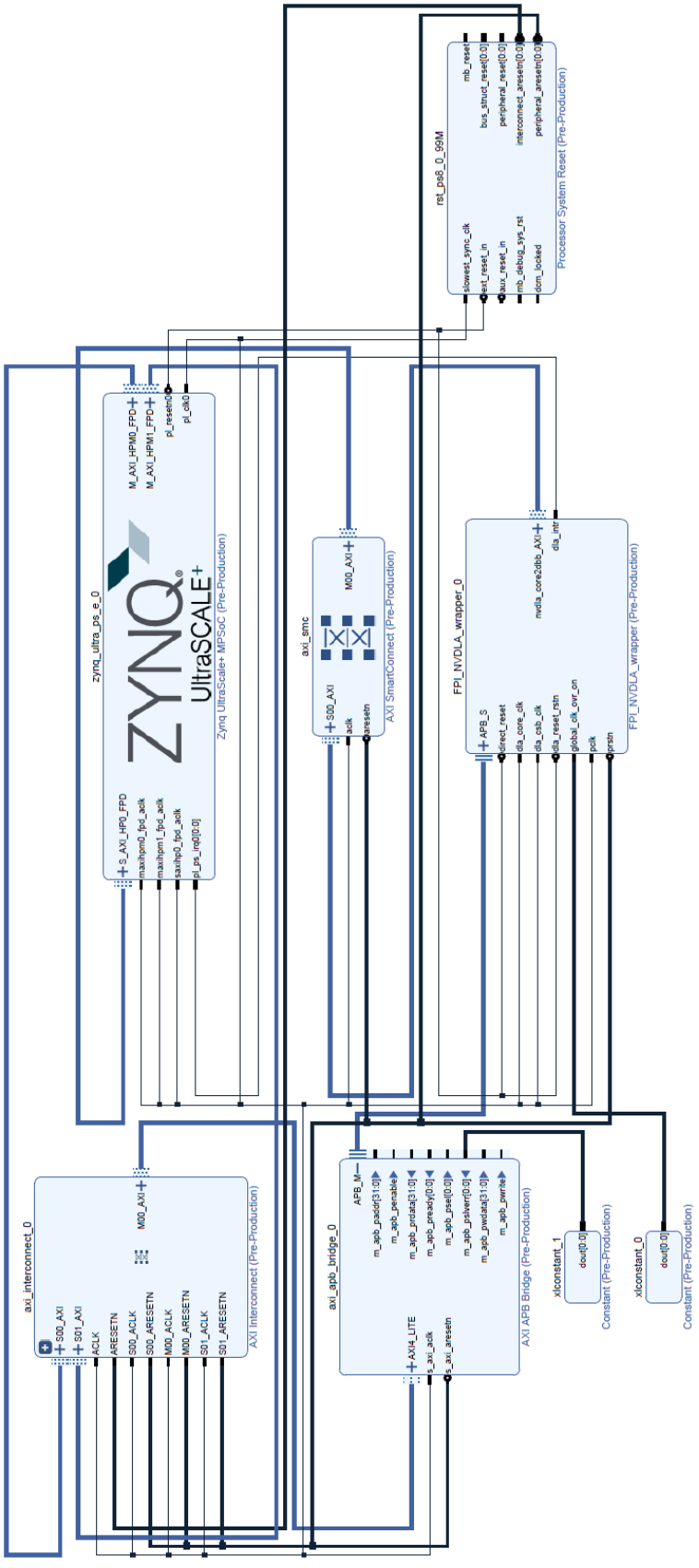


Figure 4.6: Block diagram of the top design

Before performing synthesis and implementation, another thing has to be done: in order to implement the NVDLA on FPGA, there is the need to set the Verilog define **FPGA** equal to 1. If this variable is not set, the RTL during synthesis will implement some Black Box structures that the implementation process will be not able to solve. Vivado allows to set this variable in a very simple way.
After setting this define, the steps to follow are the same performed for the wrapper:

- Generate the output products (again, with the *Out Of Context* option);
- Create an RTL description of the complete block diagram;
- Synthesis and implementation, on which a focus will be done in the next sections.

4.6 Synthesis

The synthesis, in this case, is fundamental in order to perform a first timing and resource analysis.
Different syntheses have been performed, considering different clocks and different synthesis options, in order to understand what is the maximum frequency at which the NVDLA can work on the ZCU102 board.

4.6.1 Resource analysis

Anyway, the first thing to look at is the resource report generated after the synthesis, which shows the quantity of resources that the NVDLA occupies. Of course, the allocation of the resources can change if different syntheses are performed, in particular considering different frequencies. However, the percentage of used resources is more ore less the same, so there is no need to report all the resource reports obtained with different frequencies. A glance at the allocation of the resources is showed in table 4.1.

RESOURCE	UTILIZATION	AVAILABLE	Utilization %
LUT	80959	274080	29,54
LUTRAM	689	144000	0,48
FF	94802	548160	17,29
BRAM	100,50	912	11,02
DSP	32	2520	1,27
BUFG	1	404	0,25

Table 4.1

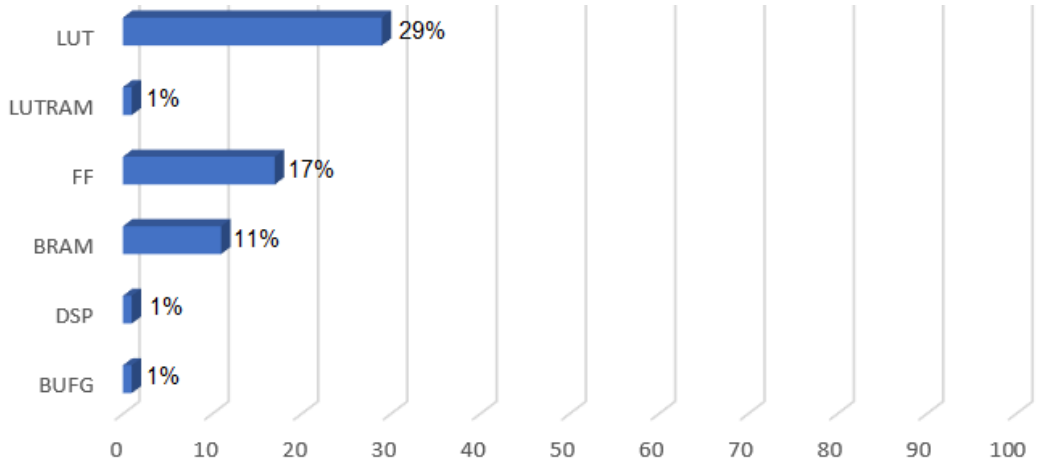


Figure 4.7: Resource utilization in percentage of the NVDLA small

4.6.2 Timing analysis

Timing analysis is fundamental in order to understand the maximum frequency at which the NVDLA can work on the Zynq Ultrascale+. Difference frequencies have been considered, starting from a very relaxed time and decreasing the period until the point at which the slack becomes negative and the timing is not closed.
The first synthesis has been generated with a frequency of 10 MHz (that corresponds to a period of 100 ns). This frequency is very relaxed, the slack is positive and the timing constraints are met. Table 4.2 shows all details about timing analysis.

Frequency	Worst pulse width slack	Worst negative slack	Total negative slack
10 MHz	48,498 ns	43,449 ns	0,000ns

Table 4.2

Slack is the difference between the expected arrival of a signal and the actual arrival of a signal. A signal should reach its destination before its expected arrival. In case of a flip flop, the signal should arrive before the next-rise edge of the clock. If the signal arrives after its expected arrival, the slack is negative and the design fails.
In this case, the **worst pulse width slack** is the difference between the rising edge (50 ns) and the **critical path**, that is assumed to be 1,502 ns. In this case, the time of the critical path is quite small, so this does not create problems if the frequency is increased. A bigger problem is represented by the worst negative slack, that is the slack on the setup time. This decreases dramatically when increasing the frequency. When this value is negative, the system is not supposed to work anymore. In order to understand the reason of this change, it is useful to introduce the concept of **speed grade**. Originally, speed grades for FPGAs

represented the time through a look up table; now they give an idea of the "speed" of the FPGA. Speed grade can affect the setup and hold time, and then the worst negative slack. The ZCU102 board has a speed grade of -2. If the speed grade increases (decreases in absolute value) the FPGA is faster. So it happens that two FPGAs with a different speed grade can run the same design with different maximum frequency.
Then, the frequency has been increased to 20 MHz (50 ns). With this frequency the critical path is the same, but the worst negative slack decreases: anyway, it is still positive and very high, and the timing constraints are met, as it is possible to observe in table 4.3.

Frequency	Worst pulse width slack	Worst negative slack	Total negative slack
20 MHz	23,498 ns	16,487 ns	0,000ns

Table 4.3

Other tentatives were made increasing the frequency to 50 MHz (25ns) and 75 MHz. The Worst negative slack becomes smaller and smaller, up to the point at which it is very close to 0. This suggests that increasing the frequency the slack becomes negative, and this actually happens at the frequency of 100 MHz. Table 4.6 shows a summary of the timing analysis with the different frequencies, up to 100 MHz.

Frequency	Worst pulse width slack	Worst negative slack	Total negative slack
50 MHz	23,498 ns	3,371 ns	0,000ns
75 MHz	8,498 ns	0,561 ns	0,000ns
100 MHz	3,498 ns	-0,404 ns	-3,704 ns

Table 4.4

Looking at the timing analysis it can be observed that the maximum frequency at which the NVDLA can work on the Zynq Ultrascale+ FPGA is 75MHz.

4.7 Power analysis

A little glance at the power consumption is due to be considered. Power consumption increases, as expected, when the frequency increases, even if the variation is not so high. Of course, two different sources of power have to be considered: the static and the dynamic power. The last one is an indication of the power consumption that is present only when the device is working fully. The table 4.5 shows a summary of the power consumptions at different frequencies.

Frequency	Total Power	Static Power	Dynamic power
10 MHz	3,975 W	0,726 W (18%)	3,248 W (82%)
20 MHz	4,037 W	0,727 W (18%)	3,310 W (82%)
50 MHz	4,221 W	0,728 W (17%)	3,493 W (83%)
75 MHz	4,38 W	0,729 W (83%)	3,651 W (17%)

Table 4.5

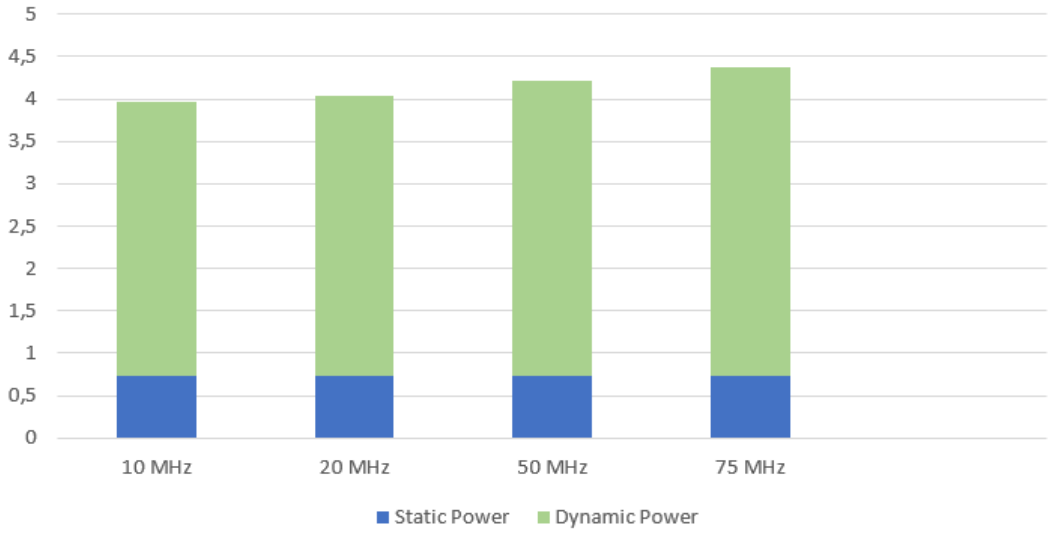


Figure 4.8: Power consumption of the NVDLA at different frequencies

A complete discussion about power consumption will be carried on in the results' chapter

4.8 Implementation

The implementation is the step which follows the synthesis. At this point, place and route is performed, trying to assign the different elements of the NVDLA to the resources provided to the NVDLA. Eventually, some timing optimizations are performed during this step.

4.9 Generation of the bitstream and of the hdf file

After having completed the implementation, the following step consists in the generation of the **bitstream**. It consists in a file containing the programming informations for the FPGA, and it represents the overall design with all its functionalities. The bitstream has to be loaded into the FPGA, but this is not a direct process: in order to proceed to the software implementation, another file has to be generated, the **hardware platform specification**

format (HDF) file. It is a sort of package containing different files, including the bitstream. In order to generate this file another tool is needed: **SDK**, that is a software of the Vivado Development Tools environment. The bitstream is exported from Vivado simulator and SDK can be opened directly from there: once a new project is created, the HDF file is generated automatically.
The HDF file is all what it is needed to move to the software implementation.

4.10 From "small" to "large" architecture

From the previous sections it can be noticed that the NVDLA occupies *just* the 29% of the total LUTs that the Zynq Ultrascale+ has. This is suggesting that it could be possible to increase the dimensions of the accelerator, increasing for example the number of MACs and introducing some functionalities that are not present in the SMALL architecture.
However, a synthesis has been performed also considering the FULL architecture. The result that has been obtained exploits that the difference between the FULL architecture and the SMALL architecture is huge, because the former requires a number of LUTs which is 10 times higher with respect to the number of LUTs available in the FPGA. A good trade-off should be to consider an architecture which is between these two extrema.

There is the possibility of generating a bigger version of the NVDLA by increasing the number of resources and the size of the convolutional buffer.
The first architecture is generated with a bigger convolutional core size. In particular:

- the "atomic size K" is the same of the initial version, so it is equal to 8;
- the "atomic size C" is increasing from 8 to 32;
- the convolutional core size is increasing from 64 to 256 (K x C);
- the convolutional buffer is the same of the initial version, and it is 128 kB wide (32x32x128);
- the architecture is still "small", so there is not the presence of the secondary memory interface, and the Rubik engine and the BDMA are still disabled.

In this way, the number of resources to be used by the FPGA is increasing, and the details about the resource utilization is specified in table 4.6:

RESOURCE	UTILIZATION	AVAILABLE	Utilization %
LUT	105177	274080	38,37
LUTRAM	961	144000	0,67
FF	117844	548160	21,50
BRAM	159	912	17,43
DSP	32	2520	1,27
BUFG	1	404	0,25

Table 4.6

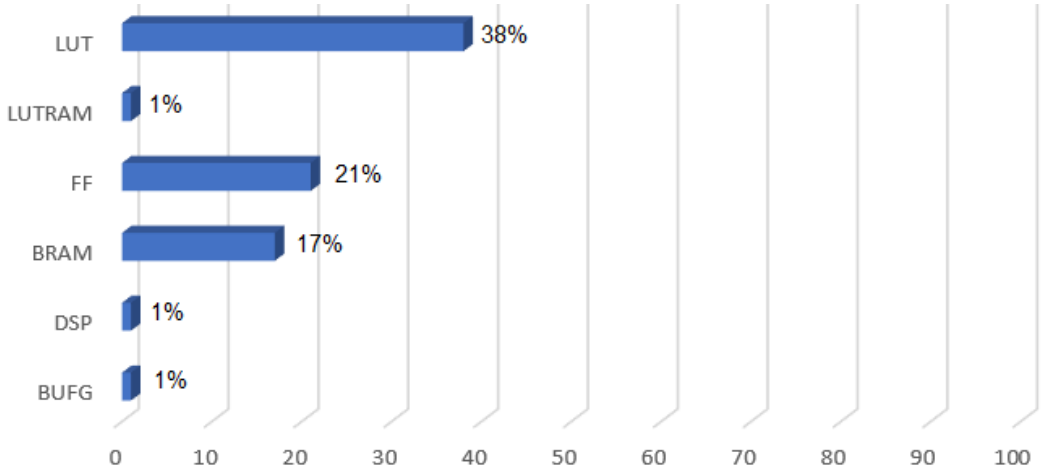


Figure 4.9: Resource utilization in percentage of the NVDLA small 256

The most important value refers to the LUT percentage utilization, that is now of the 38%. This number is quite bigger than the one of the small architecture, but it suggests that the architecture can still be increased to fit the FPGA.
For this reason, another tentative has been made with a new architecture provided by NVIDIA. This architecture shows the following characteristics:

- the "atomic size K" is increasing from 8 to 16;
- the "atomic size C" remains equal to 32;
- the convolutional core size is increasing from 256 to 512 (K x C);
- the convolutional buffer size is increasing to 512 kB (32x32x512);
- the secondary memory interface is still absent, as well as the Rubik engine and the BDMA.

The details about the resource utilization of this architecture are shown in table 4.3:

RESOURCE	UTILIZATION	AVAILABLE	Utilization %
LUT	164898	274080	60,16
LUTRAM	934	144000	0,65
FF	163686	548160	29,86
BRAM	193,50	912	21,22
DSP	65	2520	2,59
BUFG	1	404	0,25

Table 4.7

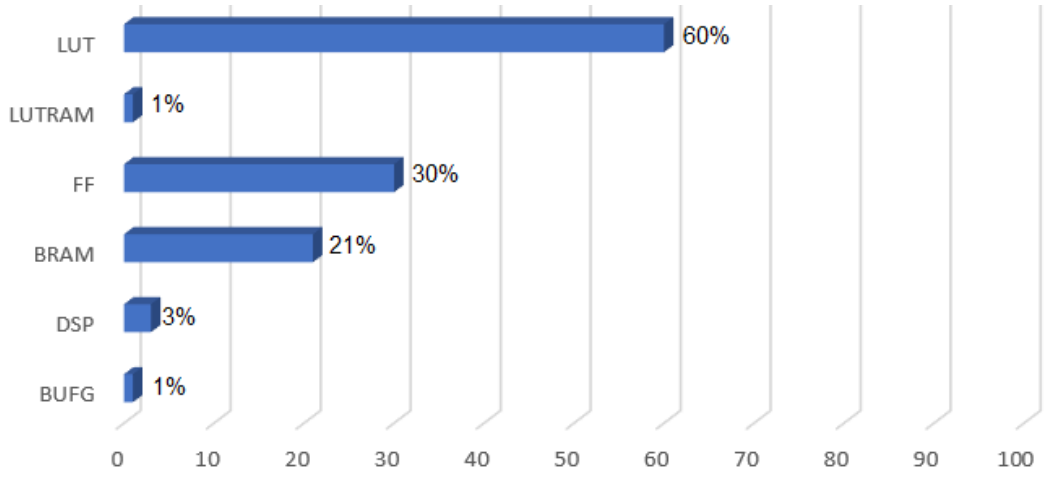


Figure 4.10: Resource utilization in percentage of the NVDLA "large" 512

The LUT percentage utilization is now bigger that the 60%. There could be the possibility of increasing the size of the convolutional buffer. Unfortunately, from the github folder it is possible to retrieve only these two architectures before considering the full one, that, as said in the previous section, is too big to fit the FPGA.

CHAPTER 5

Software implementation

This chapter describes the software implementation of the system. An introduction will be done on the software used for this part of the project and then the implementation flow, consisting in the generation of the device drivers and the development of the modules to be added to the Linux kernel will be described. Finally, the runtime application used to test the correct behavior of the NVDLA will be explained

5.1 NVDLA software development

In the previous chapter a brief introduction was given to the software part of the NVDLA. In this section, this will be treated in more details. As already said, the NVDLA has a full software ecosystem including support from compiling network to inference. [Figure 5.1]

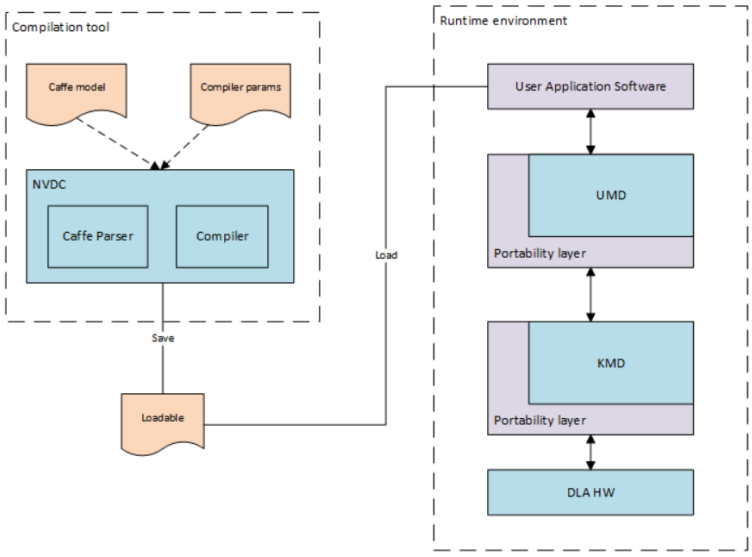


Figure 5.1: NVDLA software environment [22]

The loadable file, generated from the compiler, is used in the User Mode Driver, a runtime environment that loads this loadable and submits inference jobs to the Kernel Mode Driver. The KMD main entry point receives an inference job in memory, selects it among the multiple available jobs for execution, and submits the job to the scheduler. This core engine scheduler is responsible for handling interrupts, allowing to organize the complete flow of the data along the different functional blocks. This is useful because in this way the different engines can be used in different ways and in different moments of the execution, allowing to generate full custom layers and networks [22].
In order to reduce the latency, NVDLA introduces the concept of **ping-pong mechanism** for register programming. Each NVDLA engine has two groups of registers: **group 0** and **group 1**. The first group on which the engine is working is the group 0; while the subunit is executing, the CPU can program the group 1 in background, setting the enable bit of this group when it is done. When the execution of the processing of the hardware layer terminates on the first group, the "enable" bit of this group is cleared, and the CPU switches to the second group, that becomes active and can process the following hardware layer. Then, the process repeats, with the CPU that prepares the group 0 in background to process another layer, and so on. This mechanism allows switching between active layer without wasting cycles for configuration purposes [22].

5.2 Petalinux tool

As already said in the part dedicated to the hardware implementation, the hardware description of the NVDLA has to be loaded on an SD card in order to be injected in the FPGA. However, the HDF file generated by SDK is not the only one needed to load correctly the hardware design on the FPGA. Other files are needed, like the **Image** file containing all the **Board Support Package** of the Zynq Ultrascale+ and a Linux OS on which the board has to run. In order to generate these image files, Xilinx provide a very useful automatic tool: **Petalinux**. Petalinux is a Xilinx development tool that contains everything necessary to build, develop, test and deploy Embedded Linux systems. It consists on pre-built binary images, a fully customizable Linux kernel for the Xilinx device that is used, and a Software Development Kit that automatically configures, compiles and deploys the system that has to be loaded on the FPGA.

5.3 Petalinux flow

The version that has been used for the purpose of the thesis is Petalinux 2017.4. This version contains all the source files of the **Linux Kernel 4.9**, that has been adapted for the Xilinx

architectures, since it contains some modules related to the Xilinx boards.
The first thing to do is to create the new project and load the HDF file generated by Xilinx SDK, containing all the implementation informations about the system composed by the NVDLA and the Zynq Ultrascale+. Two instructions have to be executed:

```
$  petalinux-create --type project --template zynqMP --name
NVDLA_zynq
```

This first instruction is useful to create a new project, give to it a proper name and assign the corresponding BSP: the *template* option, for the purpose of the thesis, has been put to **zynqMP**, that corresponds to the Zynq Ultrascale+ BSP.
Then the second instruction can be executed:

```
$  petalinux-config --get-hw-description=/path/system.hdf
```

This instruction is executed in the folder of the new project that has just been created, and it is useful to import the hardware configuration. In this way PetaLinux software platform is ready to build a Linux system, customized to the hardware platform.
At this point the project can be modified, working on the Linux kernel and on the different modules that eventually have to be added to the design. This has to be done before performing the *build*.
The *petalinux-config* option can be also used to choose the Linux kernel to be configured. There is the possibility to use an internal kernel or an external kernel.
In this project the internal kernel has been chosen: the source code can be modified, and this is useful for debugging purposes.
Petalinux gives the possibility to reconfigure the kernel and to compile it. The first thing can be done by considering the instruction:

```
$  petalinux-config -c kernel
```

With this instruction a new window appears in which different options can be modified, starting from the name of the kernel. To compile the kernel, instead, the instruction:

```
$  petalinux-build -c kernel
```

can be executed.
Once the kernel has been defined, the proper modules needed to set the **device driver** related to the NVDLA can be created.
A device driver, in general, is the lowest level software which runs on a processor, since it is directly connected to the hardware implementation of the pheripheral. In this context, the kernel can be seen as an application based on different device drivers, and each device drivers is working on a part of the processor. Some device drivers are necessary to the kernel, others can be added. The ARM architecture defines a way to describe a system hardware: the **device tree**. A device tree is a tree data structure with nodes that describe the devices in a system [21]. The device tree provides informations about the cpu, the memory region, the clock frequencies used and all the peripherals. When importing the hdf from SDK, a section dedicated to the programmable logic is also defined. In particular, the NVDLA wrapper is defined in this way:

```
FPI_wrapper_NVDLA_0: FPI_wrapper_NVDLA@a0000000 {
                compatible = "xlnx,FPI-NVDLA-wrapper-1.0";
        interrupt-parent = <&gic>;
        interrupts = <0 89 4>;
        reg = <0x0 0xa0000000 0x0 0x10000>;
    }
```

The *interrupt* region is referring to the interrupt of the NVDLA, defined in the wrapper. The *reg* section, instead, defines the start address (to be considered as 0xa0000000) and the register size (0x10000), that is 64Kb. This region has been defined during the Vivado implementation. Finally, the *compatible region* is used as reference to the source code for the module insertion. This name must be the same of the one defined in the source code, so it has to be changed. In order to do this, petalinux provides a file called **system-user.dtsi**, where the elements of the device tree can be changed. For what concerns the *compatible* region, this has to be added:

```
&FPI_wrapper_NVDLA{
                compatible = "nvidia,nvdla_2";
    }
```

5.4 KMD

A device driver can be added to the Linux kernel is by inserting a **module**.
A module is a code section that is called by the kernel to communicate with the peripheral related to it. The module has an interface, constituted by a series of callbacks functions that

are called when needed.
A module must be compiled with the linux kernel, generating an object element *.ko* and loaded in the kernel with the command *insmod*.
In the case of the NVDLA, the module to be loaded in the linux kernel is nothing else that the KERNEL MODE DRIVER.

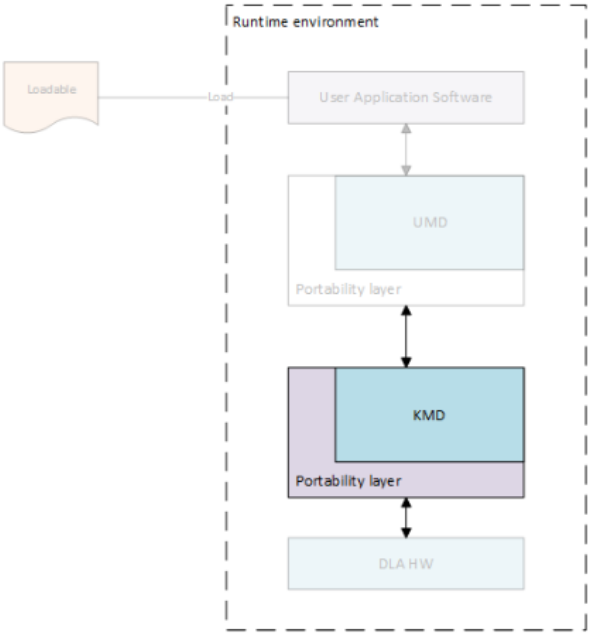


Figure 5.2: The highlighted part of the picture corresponds to the KMD section [22]

The "main" of the KMD is represented by a file called *call_callbacks.c*, that contains the **probe** method. This function is used to perform the device initialization, by initializing hardware, allocating resources, and registering the device driver. The file also contains some callback functions. Hardware is registered by relating it to the corresponding physical system defined in the device driver. The function "of match" allows to relate the probe function to the *compatible* section of the device tree.
Another important method called by the KMD is the **drm_probe**, referring to the **Direct Rendering Manager**. It is a Linux kernel subsystem that allows multiple programs to use video hardware resources together. The DRM is the only system which can communicate with the GPU, since all elements that want to use it have to ask "permission" to the DRM, that is a sort of arbiter which tries to deal with all possible conflicts. The Direct Rendering Manager is particularly important for the purpose of this thesis because it deals the part of the memory.

NVIDIA provides a precompiled version of the KMD module, but unfortunately it cannot be used for different reasons.

First of all, the version of the Linux kernel with which the module is compiled is not the same of the one used in Petalinux, so when this module is inserted in the linux kernel there is a problem of compatibility (the pre-built module is compiled with the version 4.13.3 of the kernel, while the internal kernel that has been used for Petalinux is the 4.9). This brings to a series of problems also in the source code, that could change between a version of the kernel and the another one. An example that can be provided is about the function *drm_gem_object_put_unlocked*, that has been introduced in the 4.13 version of the Linux kernel. This function must be substituted with *drm_gem_object_unreference_unlocked*, that has the same functionality and it is present in the 4.9 version.

Another change to be performed on the KMD is about the region of memory to be reserved for the DRM. The method with this functionality is *dma_declare_coherent_memory* and it is implemented in the following way:

```
dma=dma_declare_coherent_memory(drm->dev,0x40000000,0x40000000,
0x40000000, DMA_MEMORY_EXCLUSIVE|DMA_MEMORY_MAP);
if (dma) {
        err = -ENOMEM;
        goto unref;
}
```

The first two addresses of the function represent respectively the physical and the virtual address to be assigned, the third address parameter represents the size of the memory. If the settings are left as default, an error occurs, because the physical address dedicated to the DDR memory of the FPGA, defined during the Vivado implementation, goes from 0x00000000 to 0x80000000 (2 GB). To solve this problem the physical address must be changed to 0x40000000 and the size left as default (in order to reserve 1GB of space, that is enough to store everything for the NVDLA small). For what concerns the virtual memory, it has to be reserved from the system RAM. The way in which the memory can be reserved is by modifying the device tree, as it has been done for the compatible section. The way in which a memory region can be reserved is:

```
reserved-memory {
      #address-cells = <2>;
      #size-cells = <2>;
      ranges;

      reserved: buffer@0 {
         no-map;
         reg = <0x0 0x40000000 0x0 0x40000000>;
      };
   };
```

In this way, the function can be written as:

```
dma=dma_declare_coherent_memory(drm->dev,0x40000000,0x40000000,
0x40000000, DMA_MEMORY_EXCLUSIVE|DMA_MEMORY_MAP);
if (dma) {
        err = -ENOMEM;
        goto unref;
}
```

After having performed all these changes, the module is ready to be compiled. Again, also in this case, Petalinux is very helpful, because it gives the possibility of compiling any custom module to be added to the kernel. The command with which the module can be created is:

```
$ petalinux-create -t modules -name nvdla
```

A new folder is created in the Petalinux project and the source files of the modules can be inserted there. Then, in order to compile the module with the linux kernel inserted in Petalinux, another command has to be executed:

```
$ petalinux-build -c rootfs
```

Now that the module is ready, the device tree has been modified properly and the kernel has

been chosen, the complete project can be built. The command to be executed is simply:

```
$ petalinux-build
```

This command compiles the kernel, generates the system image, and builds the module *opendla.ko*. The ZCU102 board requires another format of the system image, called uImage. In order to the generate it, the following command can be executed:

```
$ petalinux-package --image -c kernel --format uImage
```

Finally, the BOOT file can be generated. It is a sort of container storing all informations about the bitstream, the board support package and the system image. It can be generated with the following command:

```
$ petalinux-package --boot --fsbl /path/images/linux/zynqmp.fsbl
--fpga /path/images/linux/system.bit --pmufw
/path/images/linux/pmufw.elf --u-boot
```

Now, everything is ready to be loaded on the FPGA. On the SD card, the BOOT, the uImage file and the module are stored, together with the compiled module **opendla.ko** . The FPGA can communicate with a PC by means of a terminal with a UART connection. The linux OS is immediately loaded when the FPGA is switched on. The first thing to do is to insert the module. The command is the following:

```
$ insmod opendla.ko
```

This command generates the following output:

```
$ opendla: loading out-of-tree module taints kernel.
$ reset engine done
$ [drm] Initialized nvdla 0.0.0 20171017 for
  a0000000.FPI_wrapper_NVDLA on minor 1
$ Module inserted correctly (dma= 0)
```

This is showing that the device has been registered correctly. At this point, the application

can be finally run.

5.5 UMD

The USER MODE DRIVER provides an Application Programming Interface (API) for processing loadable images, submitting inference jobs to KMD. This layer loads the network into memory in a defined set of data structures, and passes it to the KMD. In Linux, the function which passes data from the user mode driver to the kernel mode driver is the **ioctl()**. User Mode Driver also implements some low level functions.

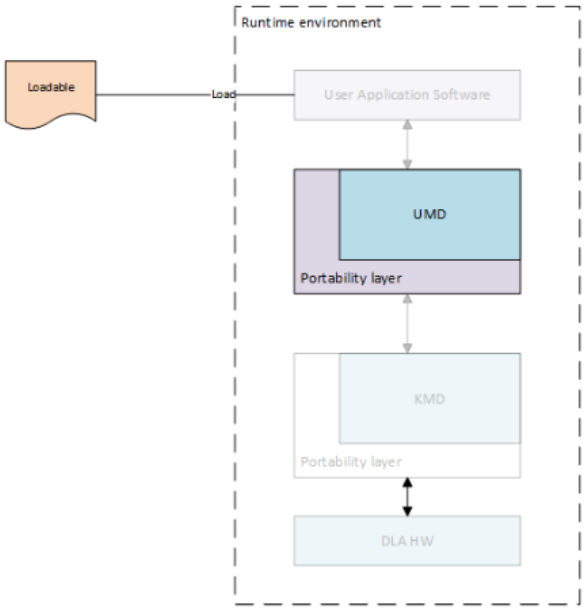


Figure 5.3: The highlighted part of the picture corresponds to the UMD section [22]

The UMD is characterized by two parts:

- The first part is about the runtime interface: at this point the weights, that are present in the pre-compiled loadable file, are stored into memory, that is allocated dynamically. The same thing happens to the input image, that is organized as an input vector before being saved.
- In the second part the job is submitted to the KMD: this is the point where the inference is really performed; the type of operation to be executed depends on the information stored in the loadable file.

The functions to access the NVDLA, allocate memory and organize the input image are implemented in the UMD as a **portability layer**: for these functionalities UMD has to

communicate with KMD and the communication interface is OS dependent.

The USER MODE DRIVER has to be compiled. This can be done by performing cross-compilation because the UMD has to be compiled for ARM. As the NVDLA website suggests, the UMD can be compiled in the following way:

```
$ export TOOLCHAIN_PREFIX=aarch64-linux-gnu
```

This instruction is used to select the cross-compiler.

```
$ export TOP=/path/output
```

This instruction is used to select the output folder on which the bin file will be generated.

```
$ make
```

With this instruction the compilation is performed.
After the compilation, a **runtime** file will be generated. At this point, the runtime file, together with a runtime library that is also generated during the compilation, is loaded on the SD card.

Once the FPGA is ready, the runtime test can be run in this way:

```
$ ./runtime --loadable regression/flatbuf
```

NVIDIA provides some pre-compiled loadable files for the NVDLA small. They allow to test all the engines separately (CONVOLUTION, NORMALIZATION, POOLING and ACTIVATION). Moreover there is one loadable file that implements the AlexNet architecture, that is a complete Neural Network thanks to which the real performances of the NVDLA can be evaluated. When a layer is executed, this message appears:

```
[...]
$[ 5461.882359] Handle cdma weight done event, processor
                Convolution group 1
$[ 5461.883182] Handle op complete event, processor Convolution
```

```
5                   group 1
$[ 5461.883953] Enter:dla_op_completion processor Convolution
7                   group1
$[ 5461.884713] Completed Convolution operation index 28 ROI 0
9 $[ 5461.885417] 4 HWLs done, totally 30 layers
[...]
```

The first four rows indicate that in this a Convolutional operation is being executed, so the Convolutional engine is invoked. Also the number of the group on which the hardware layer is processed is indicated: this number is alternating between 0 and 1 due to the ping-pong mechanism. The last row indicates the number of the hardware layers executed, and the total number of the hardware layers for the architecture stored in the loadable (in this case the architecture is the AlexNet).
When all the hardware layers are computed and the process is finished, the following message is shown:

```
$ Shutdown signal received
2 $ Work done!
$ Test passed
```

This indicates that the test has been executed correctly.

CHAPTER 6

Results and observations

This last chapter shows in detail the results obtained with the runtime test executed on the FPGA. The performances at difference frequencies will be shown, observing first the execution times of the different engines working separately, and then the ones of the AlexNet. Then, a comparison with other architectures will be done in order to understand the potentialities of the NVDLA. Finally, some observations will be done on the different versions of the NVDLA, moving from the small to the large architecture

6.1 Summary of the hardware implementation results

Before entering in details about the performances reached by the NVDLA at different frequencies, it is important to realize a summary of the hardware characteristics of each architecture, in order to understand the differences between them [Table 6.1]. This analysis has been already done in the chapter 4, when talking about the hardware implementation, but the table represented here puts all the results together, in order to have a good reference.

	UTILIZATION %			**TIMING**		**POWER**	
Frequency	LUT	DSP	BRAM	WNS	WPWS	Static	Dynamic
10 MHz	29,54	1,27	11,02	43,449 ns	48,498 ns	0,726 W	3,248 W
20 MHz	29,54	1,27	11,02	16,487 ns	23,498 ns	0,727 W	3,310 W
50 MHz	29,54	1,27	11,02	3,371 ns	23,498 ns	0,728 W	3,493 W
75 MHz	29,54	1,27	11,02	0,561 ns	8,498 ns	0,729 W	3,651 W

Table 6.1

6.2 Performances analysis

The runtime code and the way to compile it have been described in the previous chapter about software implementation. Different tests have to be performed, first of all considering the different engines separately, then working on a complete neural network architecture, the **AlexNet**.

6.2.1 Convolution engine

Before talking about the convolution engine performance, it is important to highlight a very important aspect about the pre-compiled loadable file related to the convolution. It actually processes two layers: the first one implements the convolution, while the second one implements a normalization function, invoking the normalization engine. This is due to the fact that the convolution operation does not send data back to memory when the layer has been processed, so there is a need of another layer able to do the storage of the output data. For this reason, from now on, when talking about the *time of execution of a convolution*, this will be actually referred to the time of execution of a convolution plus the time of execution of a normalization layer.
Assumed this, the table 6.2 shows all the details about the performances of the convolution engine with the NVDLA working at different frequencies. The execution time refers to the time required to perform a job, and it has been computed with some timestamps obtained using the *chrono* C library. The value referred to the **FPS (Frames per second)** is derived from the following formula:

$$FPS = \frac{1000}{Execution\_time(ms)}$$

This is a very simplified computation, that does not take into account other elements like the time to process data, anyway this could be a good reference.

Frequency	Time of execution	Frames per second (FPS)
10 MHz	4.382854 ms	22.8162
20 MHz	2.27813 ms	43.8957
50 MHz	1.021350 ms	97.9096
75MHz	0.740587 ms	135.028

Table 6.2: Table describing the performance of the convolution engine in terms of time of execution and frames per second

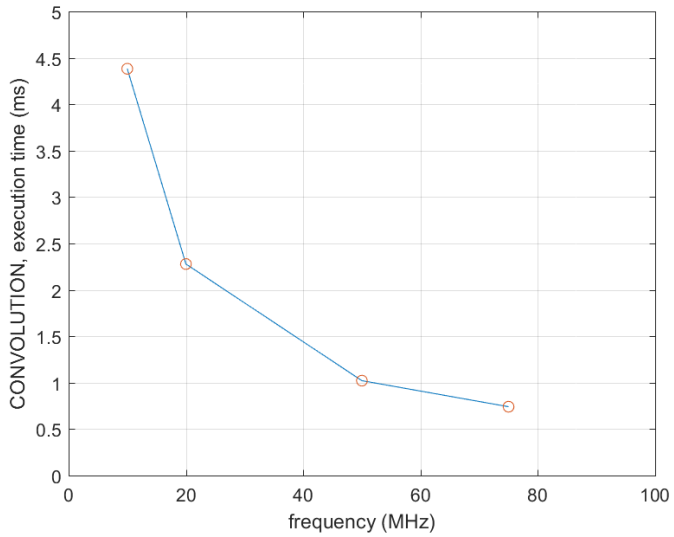


Figure 6.1: Variation of the execution time of a convolution at different frequencies

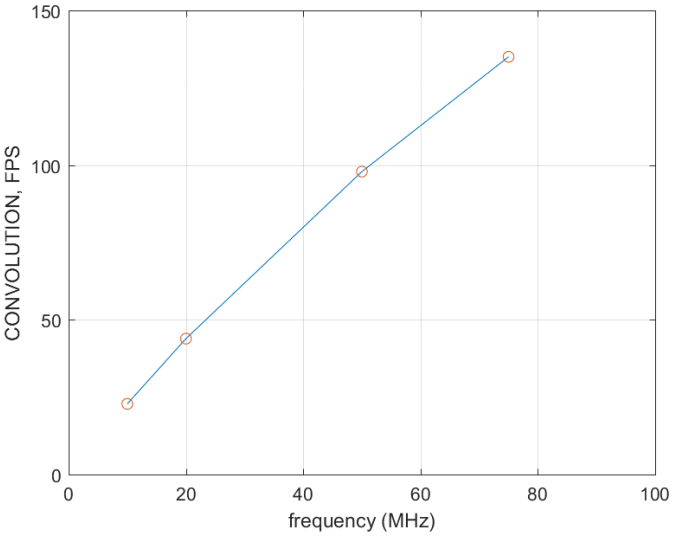


Figure 6.2: Variation of the frames per second processed for a convolution at different frequencies

Figures 6.1 and 6.2 represent, respectively, the performances of the Convolution engine in terms of execution time and frames per second. It is important to notice that the performances change linearly with respect to frequency; the FPS increase in a linear way when the frequency increases.

6.2.2 Activation engine

The same results have been obtained processing the hardware layer of the activation function. Table 6.3 shows them in details.

Frequency	Time of execution	Frames per second (FPS)
10 MHz	1.132321 ms	883,1
20 MHz	0,709038 ms	1410,4
50 MHz	0,381395 ms	2622
75MHz	0,312873 ms	3196,2

Table 6.3: Table describing the performance of the activation engine in terms of time of execution and frames per second

Figures 6.3 and 6.4 represent, respectively, the performances of the Activation engine in terms of execution time and frames per second. Also here, as for convolution, the performances change almost linearly with respect to the frequency.

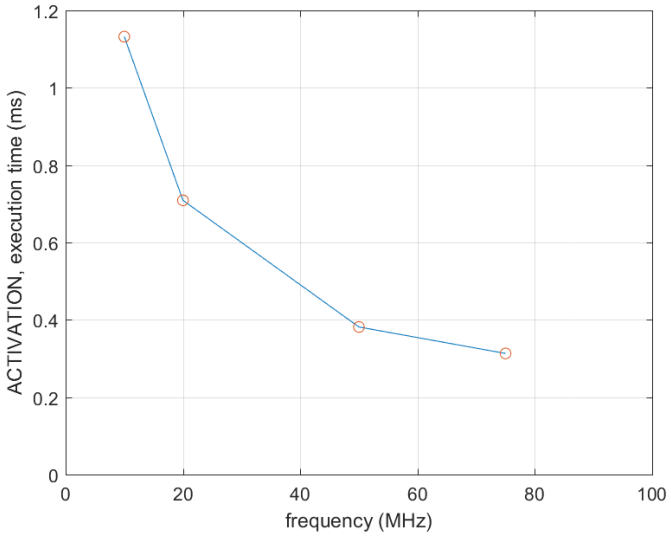


Figure 6.3: Variation of the execution time of an activation function at different frequencies

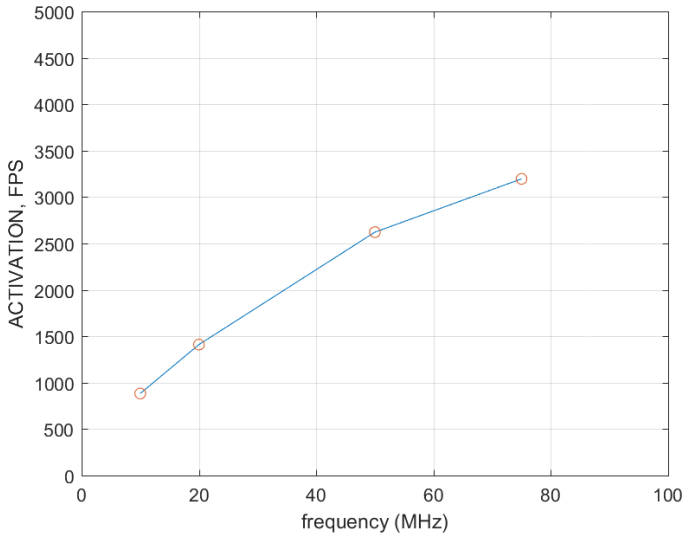


Figure 6.4: Variation of the frames per second processed for an activation function at different frequencies

6.2.3 Normalization engine

The results concerning the performance of the normalization engine are shown in table 6.4, while Figures 6.5 and 6.6 indicate that the variation of the FPS is still linear with respect to frequency.

Frequency	Time of execution	Frames per second (FPS)
10 MHz	0,500145 ms	1999,4
20 MHz	0,371304 ms	2693,2
50 MHz	0,230493 ms	4338,5
75MHz	0,164331 ms	6085,3

Table 6.4: Table describing the performance of the normalization engine in terms of time of execution and frames per second

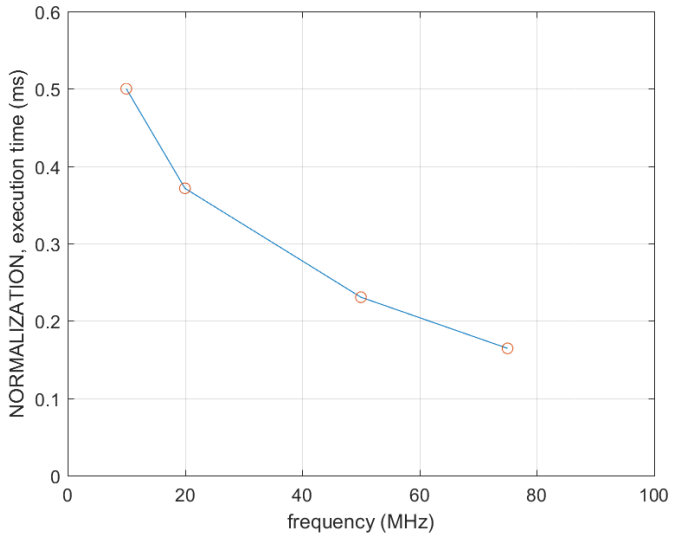


Figure 6.5: Variation of the execution time of a normalization function at different frequencies

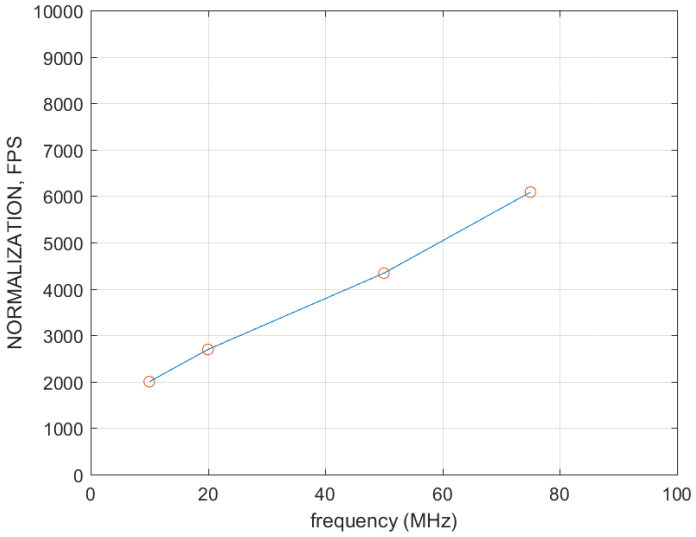


Figure 6.6: Variation of the frames per second processed for a normalization function at different frequencies

6.2.4 Pooling engine

Finally, the pooling engine is analyzed. From the table 6.5 and the following figures 6.7 and 6.8, the same behavior of the other engines can be noticed.

Frequency	Time of execution	Frames per second (FPS)
10 MHz	0,507205 ms	1971,6
20 MHz	0,376684 ms	2654,7
50 MHz	0,231863 ms	4312,9
75MHz	0,167892 ms	5956,2

Table 6.5: Table describing the performance of the pooling engine in terms of time of execution and frames per second

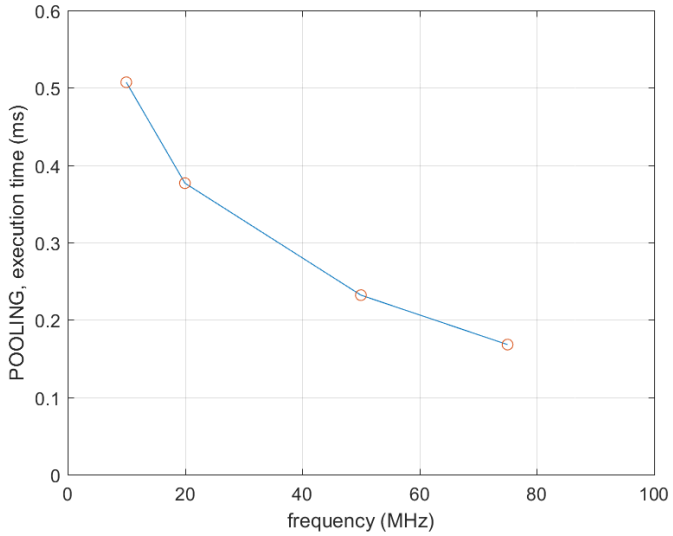


Figure 6.7: Variation of the execution time of a pooling operation at different frequencies

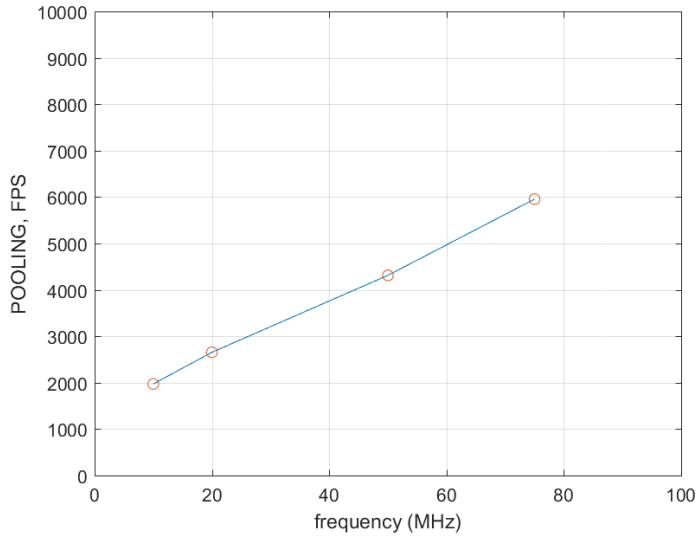


Figure 6.8: Variation of the frames per second processed for a pooling operation at different frequencies

6.2.5 Alexnet performance

The most important test has been performed taking into account a complete neural network architecture, the AlexNet (2012). This architecture processes all the layers described before, and in particular it is defined as a **thirty** layer architecture.
Before showing the details about the performance for the different frequencies, it is important to do two observations:

- the pre-compiled loadable file is impossible to read: it contains the description of the architecture but also the input image, that is not possible to see; this means that the AlexNet can receive at the input only the image given by the compiler, so it is not possible to change the image;
- the output file that is generated is a binary file, so it is not possible to observe the final classification; however, the output file has been compared with the one given by the NVDLA developers, and it is the same, so the result is assumed to be correct.

Given these observations, the performances of the AlexNet architecture, in terms of execution time and FPS, can be evaluated: the approach is the same used for the computation of the performances of the single engines. Table 6.6 shows in details these performances.

Frequency	Time of execution	Frames per second (FPS)
10 MHz	2438,84 ms	0,41
20 MHz	1220,28 ms	0,8195
50 MHz	490,41 ms	2,0391
75MHz	329,17 ms	3,038

Table 6.6: Table describing the performance of the AlexNet neural network in terms of time of execution and frames per second

Figures 6.9 and 6.10 describe the behavior of the AlexNet NN with respect to frequency. It can be noticed that the behavior, as expected, is linear.

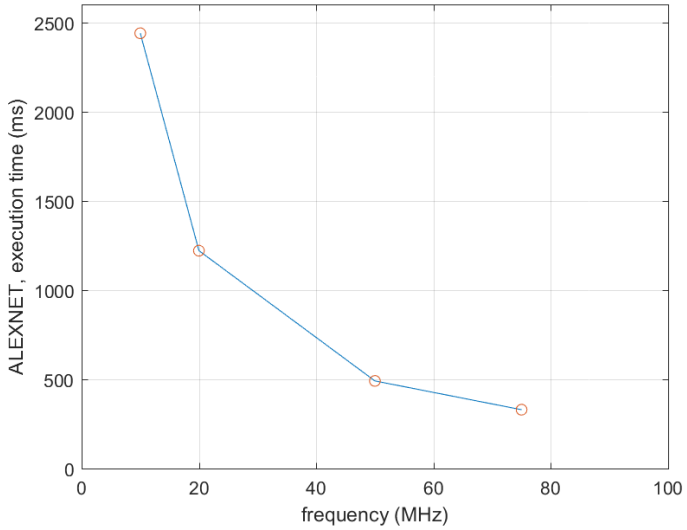


Figure 6.9: Variation of the execution time of the AlexNet NN at different frequencies

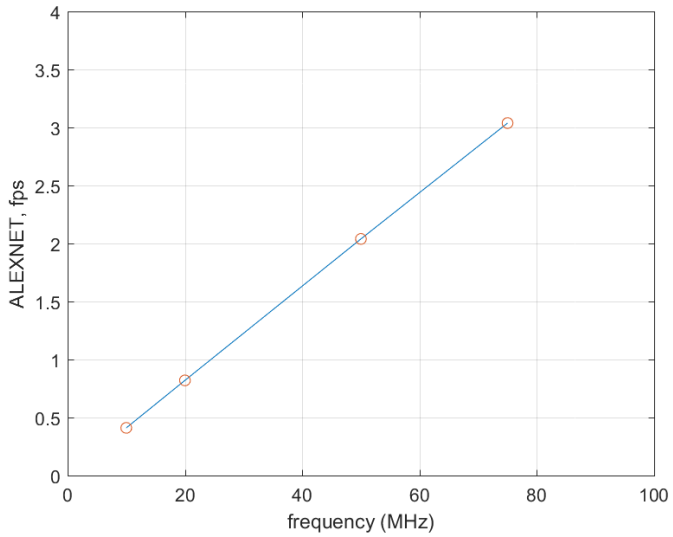


Figure 6.10: Variation of the frames per second processed for the Alexnet NN at different frequencies

Comparison of the AlexNet NN performance between the NVDLA and other architectures

The structural limits of the Zynq Ultrascale+ FPGA do not give the possibility of implementing the NVDLA working at a high frequency. Moreover, the version that has been implemented is the "small" one, characterized by a low number of MACs (64) and a low computational power. This results in performances that are not comparable with the ones of other architectures, like the GPU, that works at frequencies that are much higher, and with a much higher number of resources.
However, the linear behavior of the graph in figure 6.10 is a good reason to think that, growing in frequency, the performances of the NVDLA will be quite good. A projection can be obtained by generating the line that best fits for the available data, and then the results can be observed considering the value of that line at a point that corresponds to the frequency to be analyzed. This does not give the exact result, but at least it suggests an idea of the behavior of the NVDLA at higher frequencies, for what concerns the "small" architecture.
Using Matlab and the function *fit*, a line is generated. The line is represented in figure 6.11 and it has the following equation:

$$y = 0.0404x + 0.009441$$

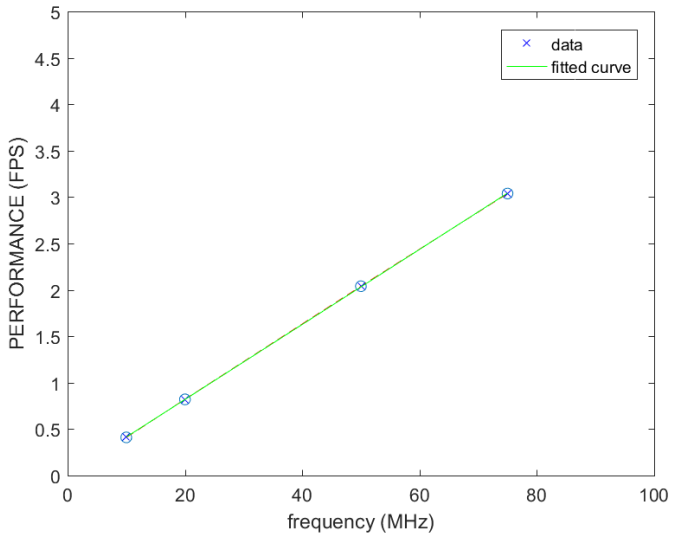


Figure 6.11: Line fitting for the points representing the performance in terms of FPS of the NVDLA, when the AlexNet is processed

From Figure 6.11 it can be noticed that the fitted line actually passes through the four points, so this line is very close to the real line. Now it is possible to find the performance in terms of FPS at higher frequencies.
The comparison was done taking into account the paper [23], where a performance analysis on inference has been done on some NVIDIA architecture and an Intel i7 core. The AlexNet is run on these architectures, and the reference frequency is **690 MHz**. Before performing the comparison, the value of the AlexNet performance of the NVDLA has to be evaluated at 690 MHz. Considering the fitted line that has been computed previously, it comes out that at 690 MHz the AlexNet is processed at **28 FPS**. Table 6.7 summarizes all details about the performance analysis on the difference architectures.

Architecture	**Performance (FPS, @690 MHz)**
NVIDIA Tegra X1	47
Xeon E52698 v3	67
Intel Core i7 6700K (FP32)	62
NVIDIA NVDLA small	**28**

Table 6.7: Table describing the performance of different architectures when the AlexNet NN is processed

The performances of the NVDLA are still lower, but it is important to consider that the

architecture is very small, with a very low number of MACs and it occupies only the 29% of the resources of an FPGA that is not the most powerful one. For this reason, these results are quite acceptable. Increasing the number of MACs, this number is expected to increase in a way that is nearly linear with respect to the dimensions of the convolutional block. Moreover, the full architecture has some engines (like the Rubik), that are able to modify the size and the dimensions of the input image, making the computation faster.
Another parameter that can be used to compare the different architectures is power consumption. In the chapter about the hardware implementation, a power analysis has been done, considering the different power consumptions at different frequencies. The variation of the power consumption with respect to frequency is shown in Figure 6.12: it can be noticed that this variation is very small, as it is expected for an FPGA.

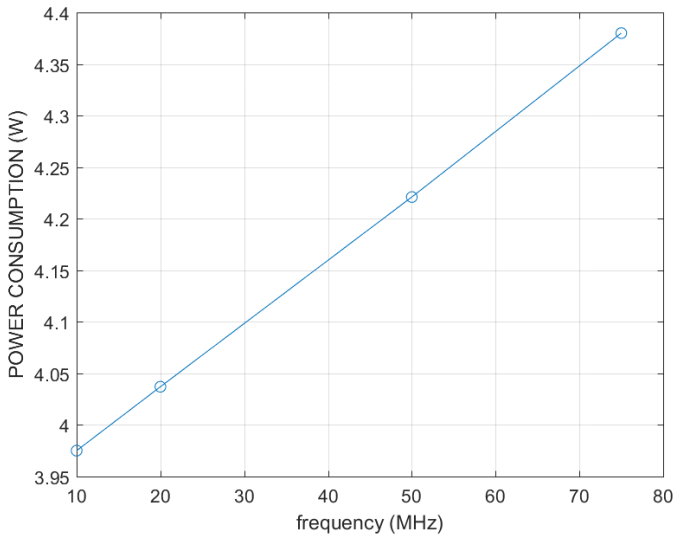


Figure 6.12: Variation of the power consumption of the NVDLA at different frequencies

Considering the power consumption, it is possible to evaluate another parameter that in general is taken as reference when analyzing the neural networks. Again, this can be evaluated only for low frequencies, but the reasoning is the same done for the performance curve. The table 6.8 shows the values of the power efficiency (FPS/W).

Frequency	Power efficiency
10 MHz	0,1032 FPS/W
20 MHz	0,2030 FPS/W
50 MHz	0,4831 FPS/W
75 MHz	0,6936 FPS/W

Table 6.8: Table describing the power efficiency reached by the NVDLA at different frequencies

The line fitting for the points indicated in table 6.8, computed in Matlab, is represented in figure 6.13 and has the following equation:

$$y = 0.009091x + 0.01845$$

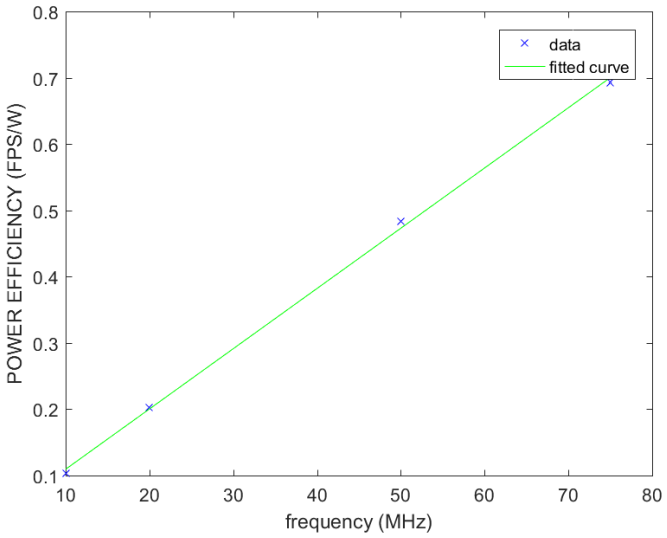


Figure 6.13: Line fitting for the points indicating the power efficiency reached by the NVDLA at different frequencies

Projecting this line at 690 MHz, the power efficiency that is obtained is **6,3 FPS/W**. The table 6.9 shows the comparison between the power efficiency of the different architectures already analyzed before.

Architecture	Power efficiency (@690 MHz)
NVIDIA Tegra X1	8,6 FPS/W
Xeon E52698 v3	0,7 FPS/W
Intel Core i7 6700K (FP32)	1,3 FPS/W
NVIDIA NVDLA small	**6,3 FPS/W**

Table 6.9: Table describing the comparison about the power efficiency between the different architectures

The power efficiency of the NVDLA is much bigger than the one Xeon and the one of the i7 core, and almost equal to the Tegra X1 GPU. From the point of view of the power efficiency the results are satisfactory: moreover, it is always important to notice that the utilization of the FPGA is still very small; increasing the dimensions of the NVDLA the power efficiency could still increase, as it will be shown in the following section.

Observation about tests on different Neural Networks

The AlexNet neural network, as well as the individual engines, can be tested on the NVDLA "small" because NVIDIA provides some pre-compiled loadable files. In order to test other architectures, the compiler, that generates the loadable file starting from the Caffe model, is needed, as explained in the previous chapters. Unfortunately, the compiler that NVIDIA has released up to now can only generate loadable files for the FULL architecture of the NVDLA. A configurable compiler is still missing, and that is the reason why other neural networks can not be tested at the moment. Due to the absence of the compiler, moreover, is not possible to generate loadable files that contain only the architecture of the neural network (recall that the pre-compiled loadable also includes the input image to be analyzed): this does not allow the user to choose its own input image to be tested.
Anyway, the results obtained with the AlexNet are quite acceptable, because they show the performance that the NVDLA "small" can have on the FPGA.

6.3 From "small" to "large" architecture: result analysis

In chapter 4, the last section was dedicated to the study of architectures that are bigger with respect to the "NVDLA small". In particular, two architectures have been taken as reference: the "NVDLA 256" and the "NVDLA 512", that increase the number of resource of the FPGA that are used. Table 6.10 summarizes the most important results for the resource utilization,

comparing the different versions of the NVDLA.

NVDLA version	LUT (%)	BRAM (%)	DSP (%)
NVDLA "small"	80959 (29,54%)	100.50 (11,02 %)	1 (0,25 %)
NVDLA "256"	105177 (38,37%)	159 (17,43%)	1 (0,25 %)
NVDLA "512"	164898 (60,16 %)	193,50 (21,22 %)	65 (2,59 %)

Table 6.10: Table describing the utilization percentage of the resource with different versions of the NVDLA

Unfortunately, without the configurable compiler, it is not possible to test these bigger architectures. Another reason for which tests are not possible is that the NVDLA "256" and the NVDLA "512" have not been tested by the developers of the NVDLA and they are not stable (a lot of compilation error where found when synthesizing these architectures).
However, a good analysis can be done about the power consumption, in order to have an idea about the variation of power depending on the number of resources of the FPGA used.
It is very useful to notice that, even with a quite high increasing of the occupation of resources of the FPGA, the power consumption remains more or less the same. This concept can be retrieved from table 6.11.

NVDLA version	Frequency	Total power
NVDLA "small"	10 MHz	3,975 W
	20 MHz	4,037 W
	50 MHz	4,221 W
	75 MHz	4,38 W
NVDLA "256"	10 MHz	3.987
	20 MHz	4,062
	50 MHz	4.285
	75 MHz	4,471
NVDLA "512"	10 MHz	4.096
	20 MHz	4,204
	50 MHz	4,387
	75 MHz	4,624

Table 6.11: Table showing the power consumption of the different versions of the NVDLA at different frequencies

Increasing the dimensions of the NVDLA means increasing the number of MACs and the size of the convolutional buffer (just to recall, moving from the NVDLA "small" to the NVDLA "512", the number of MACs increases from 64 to 512). This suggests that the performances tend to increase, because the operations can be executed in parallel among the different convolutional elements: even if the performances can not be retrieved, it seems quite logical to think that, doubling the number of MACs, the execution time of the convolution operation halves, because the operations are distributed among a number of elements that is the double with respect to the initial one.
Considering this hypothesis, the Power efficiency could still increase to a value which is much bigger than any other architecture used for the comparison.

CHAPTER 7

Conclusions and future works

The main aim of the thesis was to verify the performances and the behavior of the NVDLA, implemented on FPGA. In order to obtain these results, different commercial tools, as the Vivado Tool Suite and Petalinux, were learnt and used in order to take advantages of all their potentialities.

The performances have been evaluated at different frequencies, observing the decreasing of the execution time when the frequency increased.

Unfortunately, the FPGA was quite small, so there wasn't the possibility to work with the FULL architecture of the NVDLA, that is the biggest architecture containing the maximum number of MACs, the maximum size of the convolutional buffer, some optimizations for the different engines (as explained in the previous chapters) and the secondary memory interface. However, the performances that have been obtained with the NVDLA small were quite satisfactory, considering just the 30% of the resource utilization of the FPGA. Moreover, the frequencies at which the NVDLA small has been tested were quite low with respect to the frequencies at which in general these accelerators work. The reason of this, again, is in the FPGA and its speed grade, as explained previously: a sort of projection has been done, in order to compare the performances of the NVDLA with other architectures. Of course, these are not exact results, but at least they give an idea of the behavior of the Deep Learning Accelerator by NVIDIA.

- The performances of the NVDLA at 75 Mhz, that is the highest frequency at which the accelerator works correctly on the Zynq Ultrascale+ FPGA is about 3 fps.
- The performances of the NVDLA at 690 MHz, obtained drawing the line that fits for the points obtained from the experimental data is 28 FPS. These performances are lower with respect to the ones obtained with the GPUs, for example, but the dimensions of the accelerator and the lower power consumption with respect to the other systems have to be taken into account.

In order to highlight the configurability property of the NVDLA, that is one of the most important advantages of this architecture, different versions of the accelerator have been implemented: of course, the resource utilization of the NVDLA increased, as well as performances.

The absence of the compiler, underlined in the result chapter, was a quite important issue in the development of this thesis. The performances could be evaluated only for a neural network, the AlexNet, and unfortunately there wasn't the possibility to test the NVDLA for other neural networks, like the GoogleNet. In this context, it is important to underline the future works that are possible to make with this architecture. First of all, once the compiler will be released by NVIDIA, different neural networks could be applied to the system, in order to understand their performances. Moreover, a custom image could be considered as input, as in this moment this is not possible because the pre-compiled loadable file that NVIDIA provides already contains an input image that is not possible to change. Finally, since the compiler is assumed to be configurable, it could be possible to verify the performances of other versions of the NVDLA different from the NVDLA small.

However, the 28 fps obtained with an architecture with a so small number of MACs (64) and with a so small FPGA accuracy is very satisfactory, and demonstrates that the NVIDIA Deep Learning Accelerator opens to a lot of possibilities. This is confirmed by the fact that NVIDIA has released its new version of the Jetson platform, the Jetson Xavier, that has two NVDLAs incorporated [24]. It is assumed to become the heart of the new "intelligent machines", so it is perfectly suitable for automotive applications.

So, to conclude, this thesis demonstrated the configurability and the scalability properties of the NVDLA. It has been shown that the structure can be adapted to every kind of application, and this is a novelty for these kinds of accelerators. In this context, the FPGA is perfect for this kind of application: the implementation on FPGA guarantees low power consumption. This can be useful to run networks with high power efficiency, making the FPGA a good platform to perform development and prototyping of new architectures that later will be integrated on SoC.

The NVDLA is completely custom: this is an extreme adaptation of the HW with respect to the neural network to be run. Most of the vendors of these accelerators are going through this direction.

Bibliography

[1] E. Cota et al, *Reliability, Availability and Serviceability of Networks-on-Chip*, Springer Science+Business Media, LLC 2012

[2] Altaf Ahmad Huqqania, Erich Schikutaa, Sicen Yea and Peng Chena, *Multicore and GPU Parallelization of Neural Networks for Face Recognition*, Procedia Computer Science 18, pp. 349 – 358, 2013

[3] C. Mead, *Neuromorphic electronic systems*, Proceedings of the IEEE, vol. 78, no. 10, pp. 16291636, Oct 1990

[4] Johannes Partzsch and Rene Schffny, *Network-driven design principles for neuromorphic systems*, Front Neurosci. 2015 Oct 20

[5] Catherine D. Schuman, Member, IEEE, Thomas E. Potok, Member, IEEE, Robert M. Patton, Member, IEEE, J. Douglas Birdwell, Fellow, IEEE, Mark E. Dean, Fellow, IEEE, Garrett S. Rose, Member, IEEE, and James S. Plank, Member, IEEE, *A Survey of Neuromorphic Computing and Neural Networks in Hardware*, 19 May 2017

[6] Fopefolu Folowosele, Jonathan Tapson, Mark Vismer, andRalph Etienne-Cummings, *Wireless systems could improveneural prostheses*, 2007

[7] Steve B. Furber, Fellow IEEE, Francesco Galluppi, Steve Temple, and Luis A. Plana, Senior Member IEEE, *The SpiNNaker Project*, Proceedings of the IEEE — Vol. 102, No. 5, May 2014

[8] Eustace Painkras, Luis A. Plana, Senior Member, IEEE, Jim Garside, Steve Temple, Francesco Galluppi, Student Member, IEEE, Cameron Patterson, Member, IEEE, David R. Lester, Andrew D. Brown, Senior Member, IEEE, and Steve B. Furber, Fellow, IEEE, *SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation*, 0018-9200 2013 IEEE

[9] Steve B. Furber, Fellow, IEEE, David R. Lester, Luis A. Plana, Senior Member, IEEE, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown, Senior Member,

IEEE, *Overview of the SpiNNaker System Architecture*, IEEE transactions on computers. vol. 62, no. 12, december 2013

[10] Indar Sugiarto, Gengting Liu, Simon Davidson, Luis A. Plana and Steve B. Furber, *High Performance Computing on SpiNNaker Neuromorphic Platform: a Case Study for Energy Efficient Image Processing*, 978-1-5090-5252-3/16 2016 IEEE

[11] Filipp Akopyan, Member, IEEE, Jun Sawada, Member, IEEE, Andrew Cassidy, Member, IEEE, Rodrigo Alvarez-Icaza, John Arthur, Member, IEEE, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Member, IEEE, Gi-Joon Nam, Senior Member, IEEE, Brian Taba, Michael Beakes, Member, IEEE, Bernard Brezzo, Jente B. Kuang, Senior Member, IEEE, Rajit Manohar, Senior Member, IEEE, William P. Risk, Member, IEEE, Bryan Jackson, and Dharmendra S. Modha, Fellow, IEEE, *TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip*, IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 34, NO. 10, OCTOBER 2015

[12] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, et al., *A million spiking-neuron integrated circuit with a scalable communication network and interface*, 10 April 2014

[13] Jian Cheng, Peisong Wang, Gang Li, Qinghao Hu, Hanqing Lu National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences University of Chinese Academy of Sciences, *Recent Advances in Efficient Computation of Deep Convolutional Neural Networks*, 11 Feb 2018

[14] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar et al., Google, Inc., Mountain View, CA USA, *In-Datacenter Performance Analysis of a Tensor Processing Unit^{TM}*, To appear at the 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 26, 2017

[15] *Web page of the TPU*, https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, William J. Dallyy, Stanford University, NVIDIA, *EIE: Efficient Inference Engine on Compressed Deep Neural Network*, 3 May 2016

[17] *Github repo of NVDLA architecture*, https://github.com/nvdla/

[18] *AXI reference guide*, https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[19] *AXI SmartConnect*, https://www.xilinx.com/support/ documentation/ip_documentation/smartconnect /v1_0/pg247smartconnect.pdf

[20] *AXI to APB Bridge*, https://www.xilinx.com/support/documentation/ip_documentation/axi_apb_bridge/v3_0/pg073axiapbbridge.pdf

[21] *Power.org Standard for Embedded Power Architecture Platform Requirements*, Version 1.1 08 April 2011

[22] *NVDLA runtime environment*, http://nvdla.org/sw/runtime_environment.html

[23] NVIDIA *GPU-Based Deep Learning Inference: A Performance and Power Analysis*, November 2015

[24] NVIDIA *Jetson Xavier* https://developer.nvidia.com/embedded/jetson-xavier-faq

[25] Kalray *MPPA A New Era of Processing* December 12, 2013

[26] NVIDIA *NVIDIA Tesla V100 GPU architecture. The world's most advanced data center GPU* White paper, August 2017

[27] Versace M, Chandler B *The brain of a new machine* IEEE Spectr 47(12):3037 (2010)

[28] Andrew Cassidy, Paul Merolla, John Arthur, Steve Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore Wong, Vitaly Feldman, Arnon Amir, Daniel Ben-Dayan Rubin, et al. *Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores* In The 2013 International Joint Conference on Neural Networks (IJCNN), pages 110. IEEE, 2013.

APPENDIX A

Appendix

A.1 nv_small.spec

```
#define FEATURE_DATA_TYPE_INT8
#define WEIGHT_DATA_TYPE_INT8
#define WEIGHT_COMPRESSION_DISABLE
#define WINOGRAD_DISABLE
#define BATCH_DISABLE
#define SECONDARY_MEMIF_DISABLE
#define SDP_LUT_DISABLE
#define SDP_BS_ENABLE
#define SDP_BN_ENABLE
#define SDP_EW_DISABLE
#define BDMA_DISABLE
#define RUBIK_DISABLE
#define RUBIK_CONTRACT_DISABLE
#define RUBIK_RESHAPE_DISABLE
#define PDP_ENABLE
#define CDP_ENABLE
#define RETIMING_DISABLE
#define MAC_ATOMIC_C_SIZE_8
#define MAC_ATOMIC_K_SIZE_8
#define MEMORY_ATOMIC_SIZE_8
#define MAX_BATCH_SIZE_x
#define CBUF_BANK_NUMBER_32
#define CBUF_BANK_WIDTH_8
```

```
#define CBUF_BANK_DEPTH_512
#define SDP_BS_THROUGHPUT_1
#define SDP_BN_THROUGHPUT_1
#define SDP_EW_THROUGHPUT_x
#define PDP_THROUGHPUT_1
#define CDP_THROUGHPUT_1
#define PRIMARY_MEMIF_LATENCY_50
#define SECONDARY_MEMIF_LATENCY_x
#define PRIMARY_MEMIF_MAX_BURST_LENGTH_1
#define PRIMARY_MEMIF_WIDTH_64
#define SECONDARY_MEMIF_MAX_BURST_LENGTH_x
#define SECONDARY_MEMIF_WIDTH_512
#define MEM_ADDRESS_WIDTH_32
#define NUM_DMA_READ_CLIENTS_7
#define NUM_DMA_WRITE_CLIENTS_3



#include "projects.spec"
```

A.2 Declaration of the ports in the top entity of the NVDLA

```
module NV_nvdla (
    dla_core_clk //|< i
   ,dla_csb_clk //|< i
   ,global_clk_ovr_on //|< i
   ,tmc2slcg_disable_clock_gating //|< i
   ,dla_reset_rstn //|< i
   ,direct_reset_ //|< i
   ,test_mode //|< i
   ,csb2nvdla_valid //|< i
   ,csb2nvdla_ready //|> o
   ,csb2nvdla_addr //|< i
   ,csb2nvdla_wdat //|< i
   ,csb2nvdla_write //|< i
   ,csb2nvdla_nposted //|< i
```

```
    ,nvdla2csb_valid //|> o
16  ,nvdla2csb_data //|> o
    ,nvdla2csb_wr_complete //|> o
18  ,nvdla_core2dbb_aw_awvalid //|> o
    ,nvdla_core2dbb_aw_awready //|< i
20  ,nvdla_core2dbb_aw_awid //|> o
    ,nvdla_core2dbb_aw_awlen //|> o
22  ,nvdla_core2dbb_aw_awaddr //|> o
    ,nvdla_core2dbb_w_wvalid //|> o
24  ,nvdla_core2dbb_w_wready //|< i
    ,nvdla_core2dbb_w_wdata //|> o
26  ,nvdla_core2dbb_w_wstrb //|> o
    ,nvdla_core2dbb_w_wlast //|> o
28  ,nvdla_core2dbb_b_bvalid //|< i
    ,nvdla_core2dbb_b_bready //|> o
30  ,nvdla_core2dbb_b_bid //|< i
    ,nvdla_core2dbb_ar_arvalid //|> o
32  ,nvdla_core2dbb_ar_arready //|< i
    ,nvdla_core2dbb_ar_arid //|> o
34  ,nvdla_core2dbb_ar_arlen //|> o
    ,nvdla_core2dbb_ar_araddr //|> o
36  ,nvdla_core2dbb_r_rvalid //|< i
    ,nvdla_core2dbb_r_rready //|> o
38  ,nvdla_core2dbb_r_rid //|< i
    ,nvdla_core2dbb_r_rlast //|< i
40  ,nvdla_core2dbb_r_rdata //|< i
    ,dla_intr //|> o
42  ,nvdla_pwrbus_ram_c_pd //|< i
    ,nvdla_pwrbus_ram_ma_pd //|< i *
44  ,nvdla_pwrbus_ram_mb_pd //|< i *
    ,nvdla_pwrbus_ram_p_pd //|< i
46  ,nvdla_pwrbus_ram_o_pd //|< i
    ,nvdla_pwrbus_ram_a_pd //|< i
48  );
```

A.3 NVDLA apb to csb converter

```
module NV_NVDLA_apb2csb (
   pclk
  ,prstn
  ,csb2nvdla_ready
  ,nvdla2csb_data
  ,nvdla2csb_valid
  ,paddr
  ,penable
  ,psel
  ,pwdata
  ,pwrite
  ,csb2nvdla_addr
  ,csb2nvdla_nposted
  ,csb2nvdla_valid
  ,csb2nvdla_wdat
  ,csb2nvdla_write
  ,prdata
  ,pready
  );
input pclk;
input prstn;
//apb interface
input psel;
input penable;
input pwrite;
input [31:0] paddr;
input [31:0] pwdata;
output [31:0] prdata;
output pready;
//csb interface
output csb2nvdla_valid;
input csb2nvdla_ready;
output [15:0] csb2nvdla_addr;
output [31:0] csb2nvdla_wdat;
output csb2nvdla_write;
```

```
output csb2nvdla_nposted;
input nvdla2csb_valid;
input [31:0] nvdla2csb_data;
//input nvdla2csb_wr_complete;
reg rd_trans_low;
wire rd_trans_vld;
wire wr_trans_vld;
assign wr_trans_vld = psel & penable & pwrite;
assign rd_trans_vld = psel & penable & ~pwrite;
always @(posedge pclk or negedge prstn) begin
  if (!prstn) begin
    rd_trans_low <= 1'b0;
  end else begin
    if(nvdla2csb_valid & rd_trans_low)
        rd_trans_low <= 1'b0;
    else if (csb2nvdla_ready & rd_trans_vld)
        rd_trans_low <= 1'b1;
  end
end
assign csb2nvdla_valid = wr_trans_vld | rd_trans_vld &
~rd_trans_low;
assign csb2nvdla_addr = paddr[17:2];
assign csb2nvdla_wdat = pwdata[31:0];
assign csb2nvdla_write = pwrite;
assign csb2nvdla_nposted = 1'b0;
assign prdata = nvdla2csb_data[31:0];
assign pready = ~(wr_trans_vld & ~csb2nvdla_ready | rd_trans_vld &
~nvdla2csb_valid);
endmodule // NV_NVDLA_apb2csb
```