POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

École Polytechnique Montreal

ELE8307: Le Rapport Final

Students :
Hossein Askari,
Nathan Heraief,
Alexandre Riviello

Teacher :
Jean-Pierre David

September 12, 2019

Contents

1 Introduction

In this report, we will present the overall design of our architecture. Essentially, we had to design a hardware/software solution using a SocKit board to accelerate a neural network-inspired algorithm working purely in software. A 24 hour and a 5 minute case were designed. This meant that we needed a solution that can adapt to a fast changing architecture (only software modifications are permitted) and to a slow changing architecture (hardware can be modified).

In the following sections, we will describe the changes that we made to our original design. We will then describe the new architecture. We will describe all the modules used in the new accelerators in detail. Finally, we will talk about the simulation and implementation results. We will show the acceleration obtained compared to what we were expecting from our simulations.

2 General Architecture

Previously in our **Rapport d'étape**, we proposed an architecture for accelerating a neural network. After starting to implement the proposed architecture, we faced a couple of technical issues when adding an ARM core (HPS core) to our design. Hence, we changed the previous architecture and designed a new one. In this section we will describe both architectures and we will show what changes we have made.

2.1 Original Architecture

Figure 1 illustrates the original architecture that we previously proposed in **Rapport d'étape**.

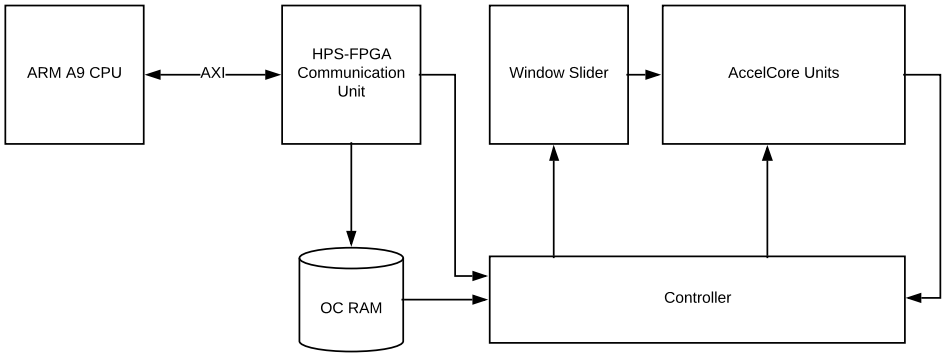

Figure 1: Original architecture proposed in **Rapport d'étape**.

As it can be seen, the original architecture consisted of 6 components. The ARM CPU, the HPS-FPGA communication unit, the on-chip RAM, the Window Slider, the controller and the AccelCore units. The idea was to run the application on the ARM CPU. Whenever we had to to use the accelerator, a command would be sent to the core through the AXI bus. The HPS-FPGA communication unit would parse the command and would perform accordingly. The commands were either loading RAMs with the weight, bias or image values or were used to start performing calculations.

In the case of performing calculations, the controller had to provide the AccelCore with the right weights and biases. Also, the controller had to signal the Window Slider to output a new valid window so that the AccelCore could start computing. By using an ASM, the controller had to keep track of the execution order and to provide correct signals and data to the AccelCore and Window Slider.

2.2 New Architecture

Figure 2 shows the new architecture. This figure shows that our new design consists of 7 major blocks. Just like before, we have a Window Slider, an AccelCore, on-chip RAM and a controller. However, in our new design, we are using two Nios and 7 special instructions.

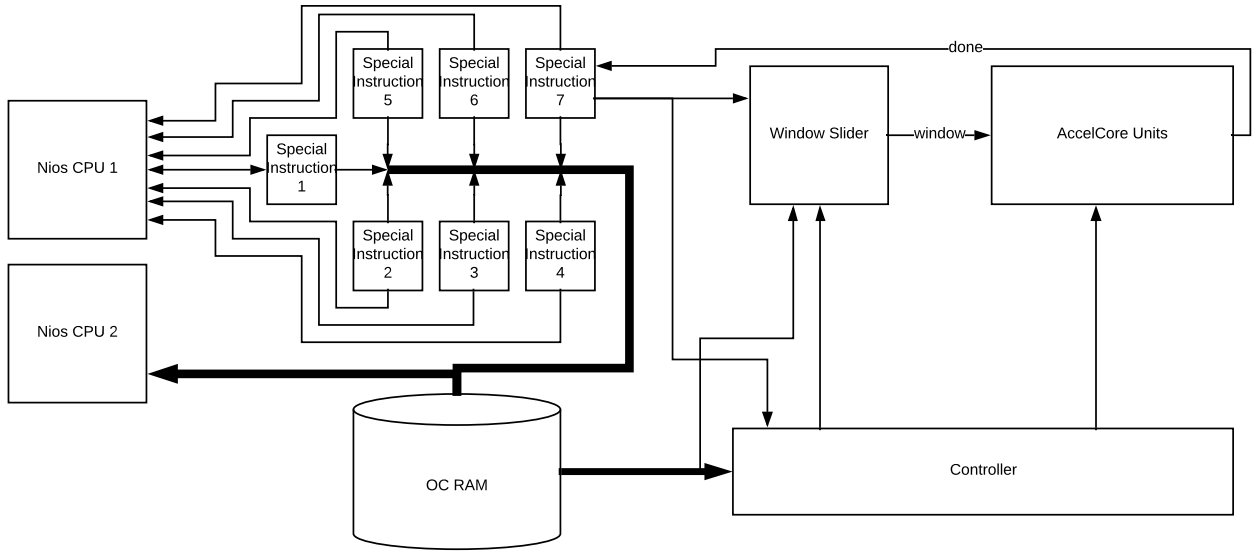

Figure 2: The new architecture. We used this architecture to accelerate the process of calculating the inference path of the neural network. We used two Nios in our design and 7 special instructions to load weights, biases and images and to send the start signal to the AccelCore.

We used two Nios in our design. The first Nios is responsible to run the main software application and to call the special instructions which will load weights, biases and images to the on-chip RAM. Also, we defined another special instruction to start the operation by giving a `start` signal to the Window Slider. This instruction is multi-cycle and it will wait to receive a `done` signal from the AccelCore. This instruction will return the result of the calculation (10 values) on the current window. On the other hand, the controller unit will use an ASM to track the state of the operation. The special instruction that gives the `start` signal to the Window Slider is also connected to the controller. This will allow the controller to know when to start to provide correct data to the AccelCore. We instantiated another Nios processor for debugging purposes.

As it can be seen in Figure 2, the second Nios is connected to all the on-chip RAMs (weights, biases and image RAM). The second Nios acts as a sniffer and it will record all transactions between the NeuralCore and the on-chip RAM. This was a great debug tool since it gave us what part of memory was accessed and what values have been written to the RAMs.

2.3 Changes and upgrades

A quick look at Figure 1 and 2 shows that the big change was in the communication module. We had to implement seven custom instructions to perform all the operations we wanted. On the other hand, we decided to use two Nios CPUs since we were not able to establish a reliable communication link between the ARM CPU and the NeuralCore. Overall, our original architecture was mostly re-used in the new version.

3 Technical Documentation

3.1 Modules

The overall architecture is composed of nearly the same blocks as the ones presented in the Rapport d'étape. Essentially, the calculations are performed by the NeuralCore. This Core is divided into many sub-blocks including a Window Slider, 3 AccelCores and 3 Line Buffers. The next component is the Memory Management, which is composed of many blocks of RAM concatenated. These were created using the Altera Wizard. Finally, the controller is the unit that takes care of giving the right weight values at the right time to the AccelCores. The full architecture is illustrated in Figure 3. The NeuralCore did not change when compared to the Rapport d'étape.

3.1.1 NeuralCore

The NeuralCore's first key component is the Window Slider. This unit is the opus of our design. Inspired by Yann Lecun's paper [1], this unit is capable of outputting a new image window every clock cycle. It works as follows: the image is placed in an array (top left element first) and everytime the slide command is called, the bits representing the image are shifted by 1 until the buffer containing the image is full. Assuming a 16x16 filter and a 60x200 image, the first 16 bits of the buffer are concatenated along with bits 60-75, bits 120-135 and so on until the entire 16x16 window is created. This concatenation represents

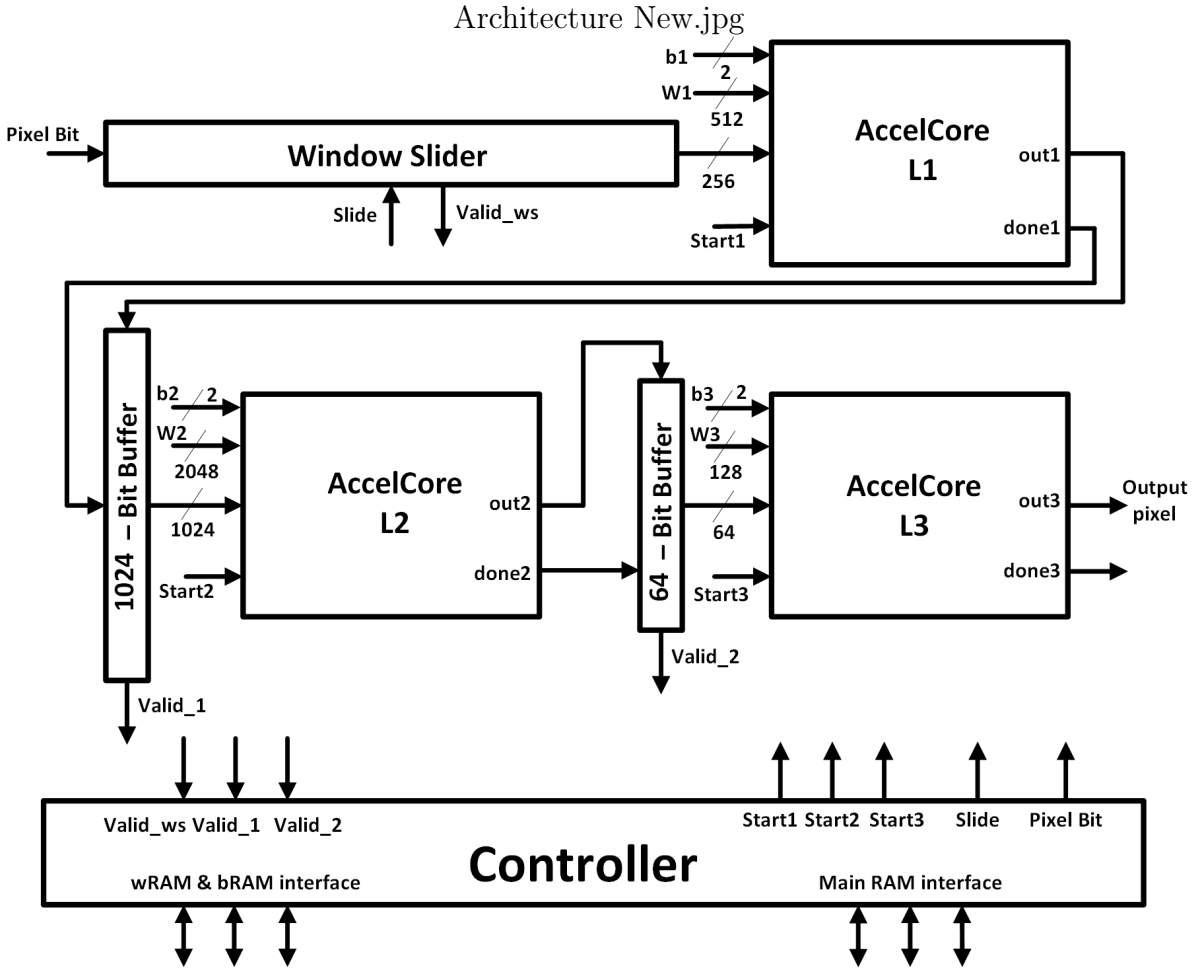

Figure 3: Full architecture of the proposed NeuralCore (With the controller).

the window of the first convolution. To obtain the second window, the Slider simply shifts all bits by 1. Figure 4 represents the operations performed by the Window Slider.

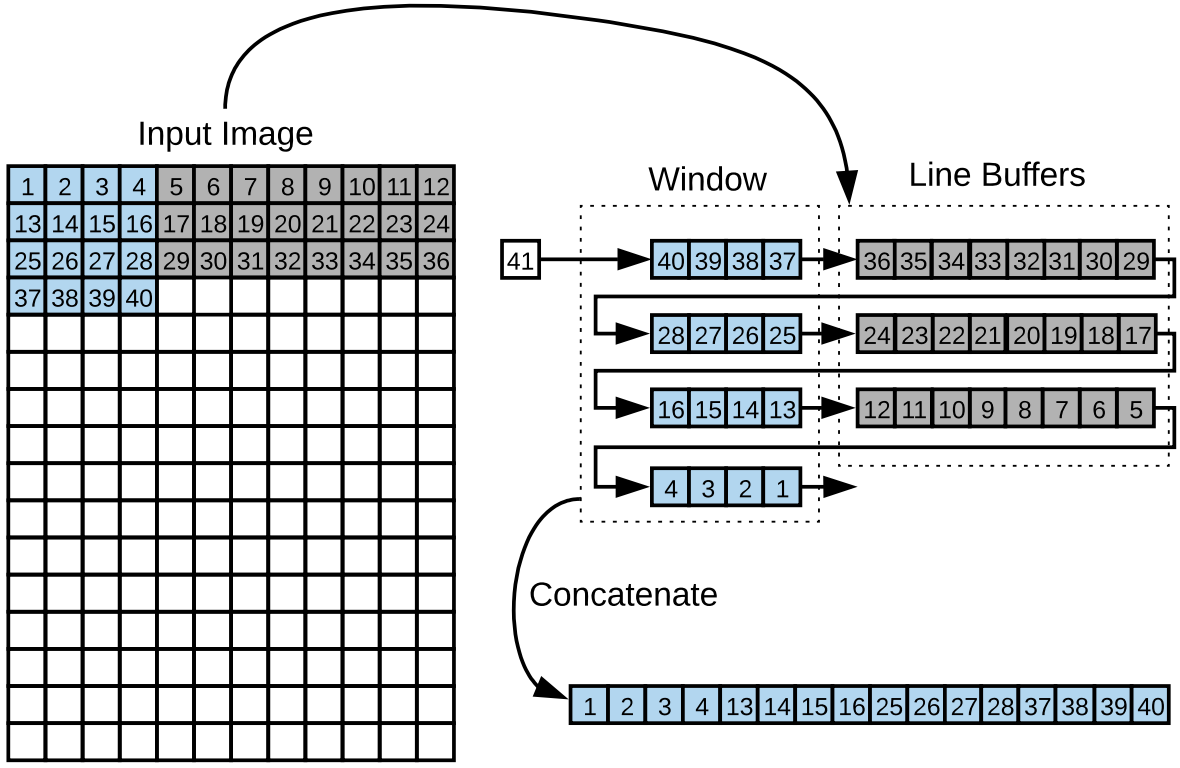

Figure 4: Window Slider module. This module is responsible to select windows from the input image and then concatenates the result to create an array which is ready to be fed to the neural net.

When the window reaches the end of the image, the slides generated are not valid until the window changes row entirely. Our state machine considers this problem and only outputs a valid signal when the window is valid. This phenomenon is illustrated in Figure 5.

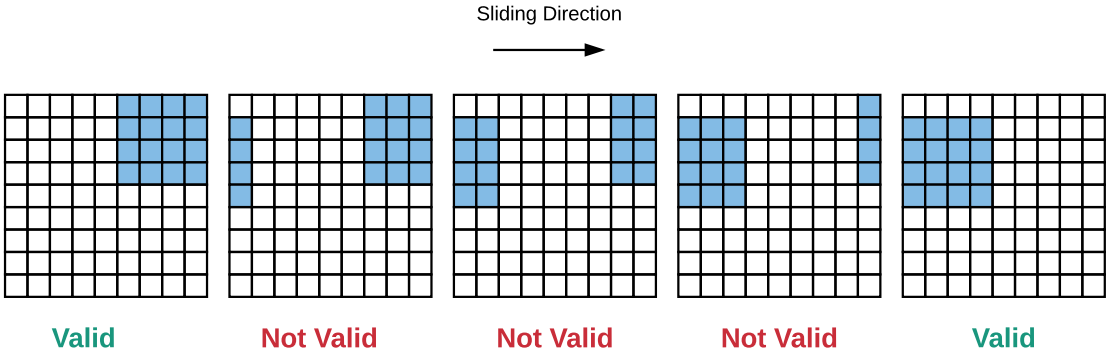

Figure 5: This image shows why not all windows are valid.

The state machine of the Window Slider is illustrated in Figure 6.

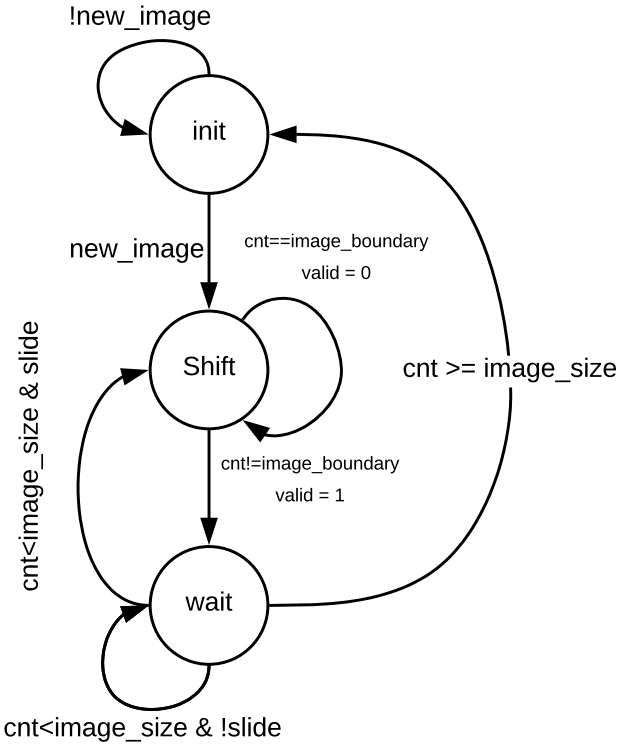

Figure 6: Window Slider ASM. The Window Slider ASM generates a valid signal only when the generated window is within the image boundary. Also, it only generates a valid window when it is requested by the `slide` signal.

Within the NeuralCore, the second most important module is the AccelCore. This is the unit that performs the MAC operations needed for convolution. Figure 7 illustrates the general architecture of the AccelCore.

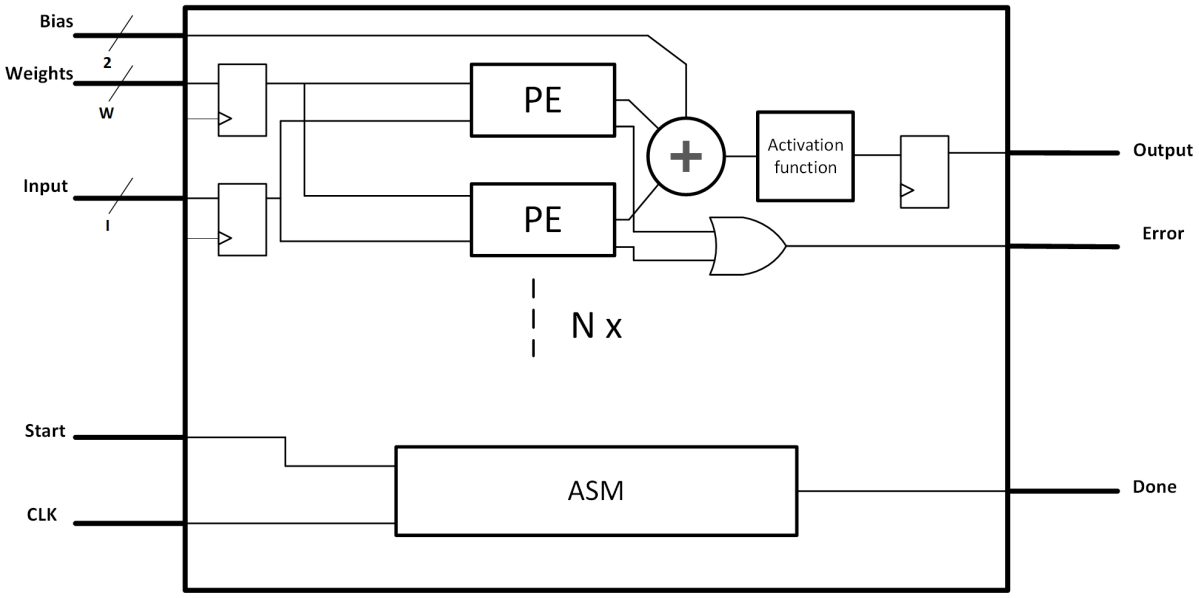

Figure 7: High-Level Representation of an AccelCore Unit.

Our design counts 3 AccelCores of different sizes. Each takes care of a single layer of the calculation. In other words, AccelCore L1 calculates the first layer (1024 neurons), L2 takes care of the second layer (64 neurons) and L3 takes care of the third layer (10 neurons). The AccelCores are pipelined to help with data synchronization and to prevent long datapaths from slowing down the clock. The calculations are performed by PEs (Processing Elements). These are illustrated in Figure 8. These PE units each perform 8x8 dot products. The weights are represented with 2 bits (ternary) and the image input is represented with 1 bit (binary). Considering the low resolution of this calculation, simple logic gates are used for every multiplication. The combinatorial logic used for a single multiplication is illustrated in Figure 9.

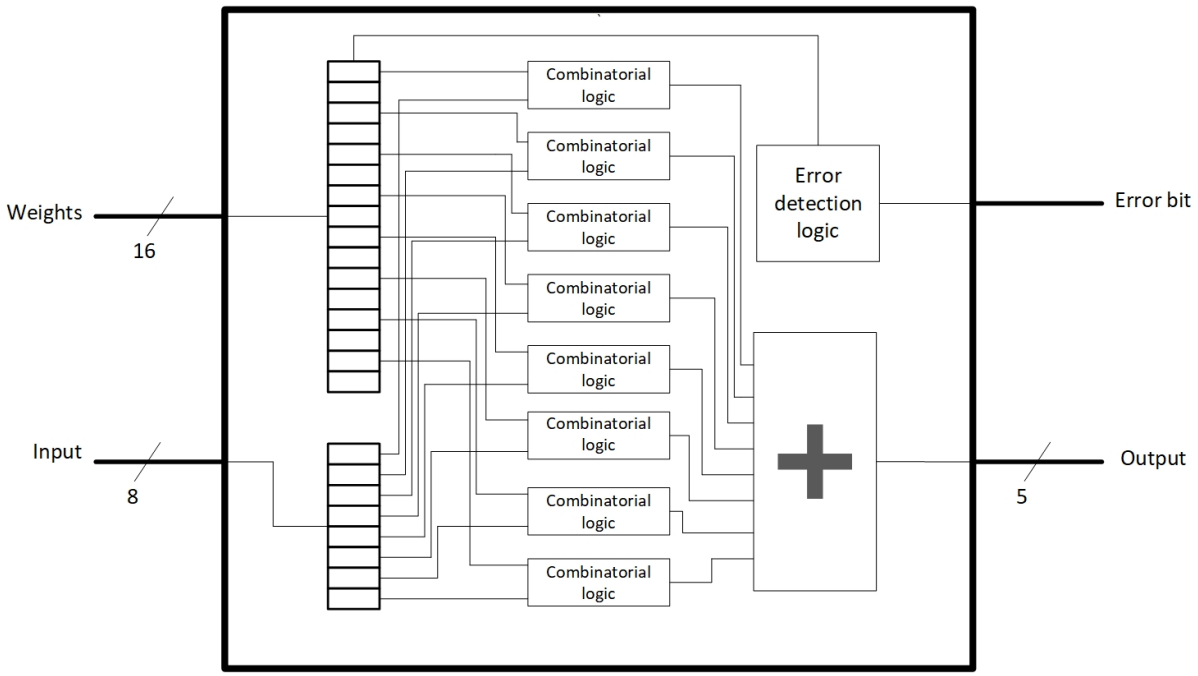

Figure 8: Representation of the Processing Element (PE).

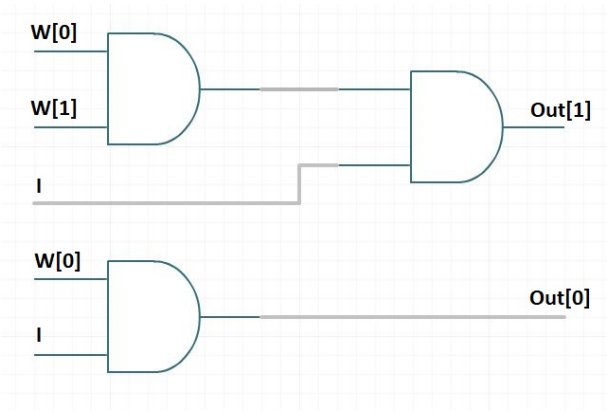

Figure 9: Combinatorial logic inside the PE.

The state machine contained inside the AccelCore is not particularly interesting. Essentially, it outputs a done signal for 1024 clock cycles (for AccelCore L1) 2 clock cycles after the start signal is triggered. This is simply to tell the LineBuffer that the output of the AccelCore is valid.

The final elements of the NeuralCore are the Line Buffers. These units accumulate the outputs of the AccelCore. It's a big shift register which shifts everytime the done signal of the AccelCore is on. After 1024 shifts (for the first Line Buffer), a valid signal is set to high for a single clock cycle. This signal is used by the controller so that it knows when the next AccelCore can start computing its results.

3.1.2 Memory Management

In order to perform the calculations, the weights, biases and image are stored inside block RAMs. In total, 7 blocks RAMs were defined (3 for the weights, 3 for the biases and

1 for the image). These RAMs use M10k blocks inside the FPGA. Since each M10k can only output 32 bits at a time, many M10ks are concatenated in parallel so that all 1024 weight values can be obtained in a single clock cycle for the first layer. The Altera Wizard automatically takes care of generating the required logic along with the M10ks. The defined RAM blocks are illustrated in Figure 10. The complete RAM block generated by Altera is illustrated in Figure 11.

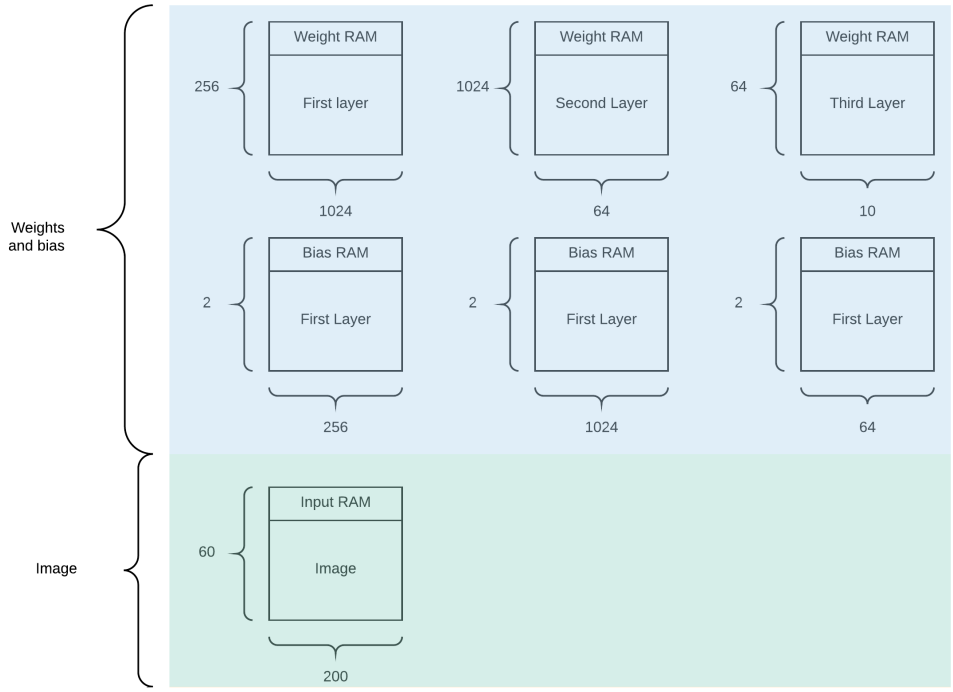

Figure 10: Memory architecture

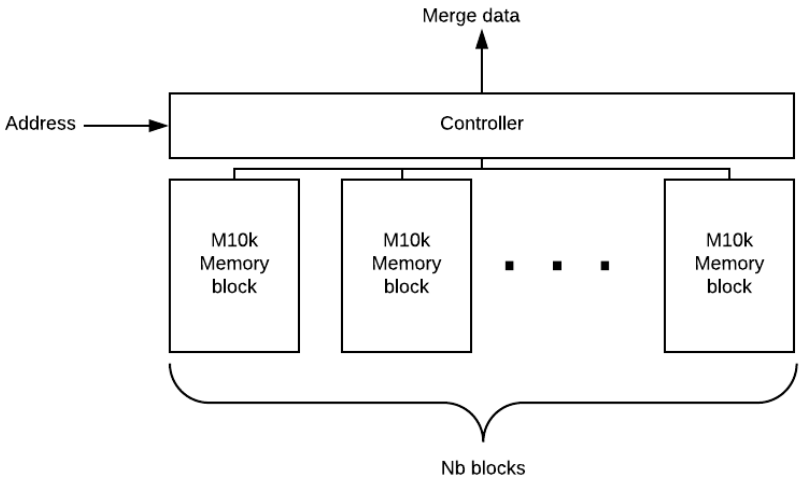

Figure 11: Data size solution

3.1.3 Controller Unit

The last module of our design is the controller unit. This unit uses the Window Slider's valid signal as well as the Line Buffers' valid signals to determine when to send the weights to the AccelCores. Once that a valid signal is activated, the unit simply outputs weights every clock cycle until it went through all the neurons. The ASM of the controller unit is illustrated in Figure 12. The controller diagram is illustrated in Figure 13.

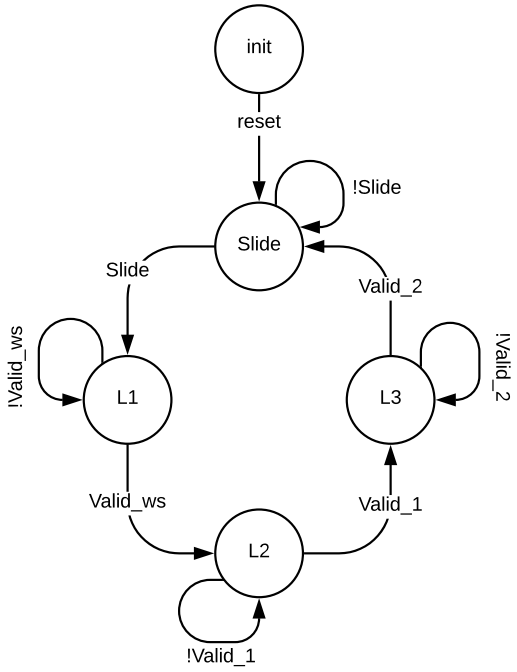

Figure 12: This figure illustrates the ASM for the proposed controller. In every layer (illustrated by L_x), the controller waits for a valid signal. This valid signal comes from the accelerator cores indicating that the computation of all neurons has been done. Based on the number of the current layer, this will trigger the controller to calculate the next address for the layer's weight RAM.

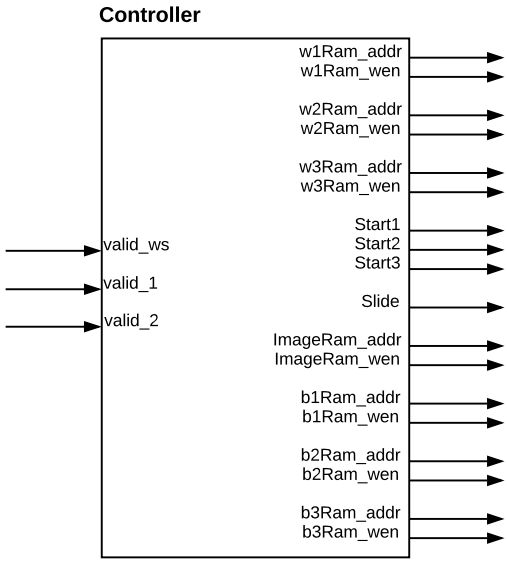

Figure 13: Controller block diagram. This figure shows the signals that are generated and consumed by the controller.

In total, there are 7 special instructions to write to RAM (3 for weights, 3 for biases and 1 for the image), 1 special instruction that calls the NeuralCore, 1 NeuralCore, 1 block containing all the RAMs, 2 Nios processors and 1 special instruction to sniff the RAMs. All these modules are connected via QSys. The connections between the NeuralCore and the memory are illustrated in Figure 14.

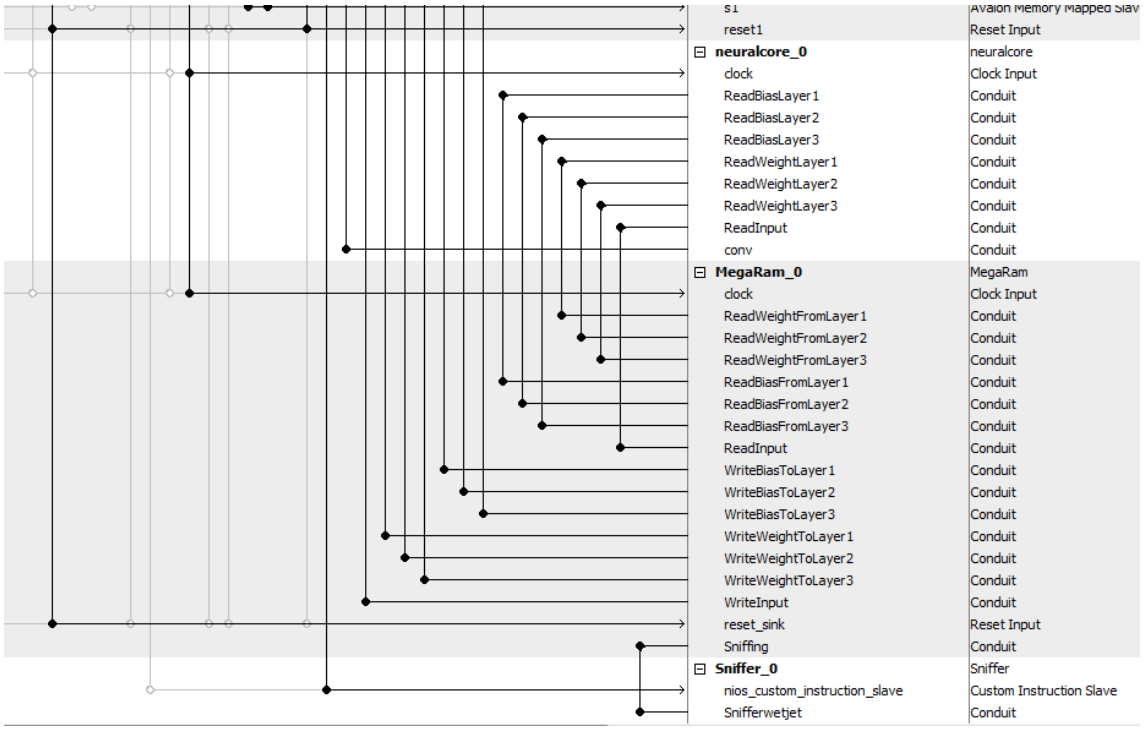

Figure 14: Qsys Diagram showing the connections between the NeuralCore and the RAM block.

3.2 Flow of Calculations

To compute the output of the neural network, the weights, biases and image values must be loaded into the RAM. This is done in software. Afterwards, a special instruction is called which computes 10 pixels (1 per image). The VGA controller then draws the pixel to the screen and the next pixel is then computed. The two main steps in this whole procedure is the loading of the data and the computing of a pixel value. The following sections detail how these steps are done.

3.2.1 Loading the data

In order to write the data to the RAM, 7 special instructions were created. The first 3 instructions write the weight values to the RAM. Since the special instructions can only take a maximum of 64 bits at a time (as input values), and that the weight vector must hold upto 2048 bits of values, it is impossible to write a full weight array in one clock cycle. Therefore, every time the instruction is called, we shift the weight values by 2 (since each weight value is 2 bits). The next time we call the same instruction, the old buffer which still contains the previous weight values shifts the old values right and concatenates the new values. Figure 15 illustrates this mechanism. When we set bit 12 to 1 of datab (input of the special instruction) the buffer is written to the RAM at the specified address. The first weights are written to address 0. The address corresponds directly to the neuron number of the respective layer. To adapt the system to the 5 minute case, the weight values that are not being used (because there are less than the maximum amount of neurons) are zero-padded. This gives our design the flexibility to adapt to any network size.

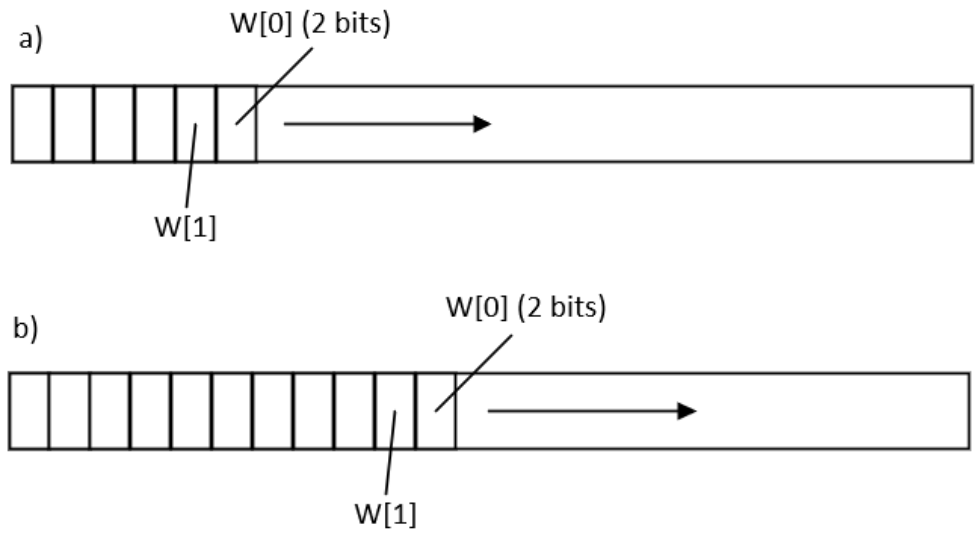

Figure 15: Populating the weight buffer. a) Filling the buffer by concatenating the new values on the left. b) Buffer after 5 extra weight values were added.

Since the biases are 2 bits, each address only needs to hold 2 bits. The special instruction simply writes the 2 bits to the specified address. Finally, every pixel of the original image is stored in a single address.

3.2.2 Calculating a pixel value

To calculate a pixel value, the flow goes as follows:

- Call the convolution instruction in software.
- The convolution instruction sends a start signal to the NeuralCore, which, in its turn, sends a start signal to the slide input of the window slider. This triggers a change in the image window.
- When the new window is ready, the Window Slider outputs a validws signal which means that the output of the Window Slider is ready for the AccelCore. The validws is also a trigger for the controller. On the next clock cycle, the controller will send the weights of address 0 (or neurone 0) to the AccelCore. From that point on, new weights, are sent every clock cycle and a new element is calculated.
- The elements that were being calculated are sent to the line buffer. The done signal of the AccelCore tells the LineBuffer to shift its result. Once the line buffer is full (after 1024 clock cycles), the valid1 signal is set to high for one clock cycle. This has the same effect as the validws signal on the Window Slider. Upon detecting this signal, the controller starts sending the weight values to the AccelCore L2 every clock cycle for 64 clock cycles.
- The process is repeated until the last AccelCore computes 10 values (which are also stored in a buffer. The final done signal is sent back to the convolution special instruction and the 10 bits representing the pixel values are sent back.
- Each bit is drawn on screen using the VGA write instruction.

3.3 Results

Finally, our design is able to produce the expected output within less than a second. The system was tested for different sizes and configurations of neural networks.

In the worst case (1024*64*10), we have an output 100% similar to the reference output given by the software solution. This allow us to validate that our result is the right one.

Figure 16: Output produced by the Neural Core

Figure 17: Output produced by the Software

The time performance of our NeuralCore is summarized in the table below (for the 24 hour case) :

	Network size	
	64*40*10	1024*64*10
Time to load the RAM (ms)	35	335
Time to calculate (ms)	17	256

To cope with a network size different than $1024 * 64 * 10$ (for the 5 minute case), we preprocess the weights. It means we padded each matrix to the max size by adding zeros. Therefore, the time to load the RAM and to compute the output is constant for every network size we can support in the 5 minute case. This time is equivalent to the biggest network size of the 24 hour case.

3.4 Specifications of our solution

We can now take a look at the specifications of our design after the implementation.

This table compares the ressources avalaible on the SoCKit board.

Ressource	Used	Avalaible	Used/Avalaible
Logic utilization (in ALMs)	15 707	41 910	37 %
Total registers	23762		
Total block memory bits	3 472 880	5 662 720	61 %
Total RAM Blocks	459	553	83 %

Our design is based on the capacity to send the 2048 bits in one clock cycle to our NeuralCore. We use the IP-Catalog of Altera to create a specialized RAM. However even if our RAM can send 2048 bits in one clock cycle it doesn't match the architecture of a M10K blocks. This is why we only used 61% of the memory bits but 83% of the RAM blocks.

3.5 Tests and Simulations pattern

Verification and validation in FPGA design is a crucial step. That's why we split the verification into following steps :

- Writing a test bench for every module (unit test)
- Writing a System level test bench to verify the integration
- Writing software models to test the simulation results

This is summarized by the Figure below :

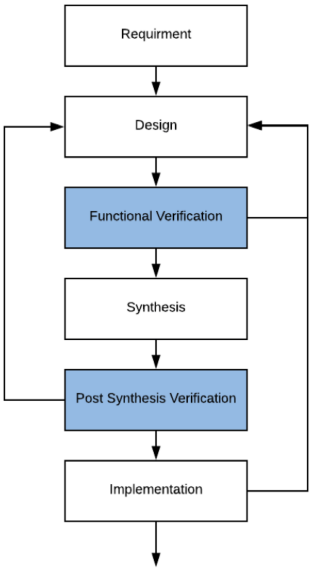

Figure 18: Verification pattern

We wrote a testbench for every modules of our solution. We also use a lot of scripts to simulate our NeuralCore and each module. The way we use scripting to debug is described in the figure bellow :

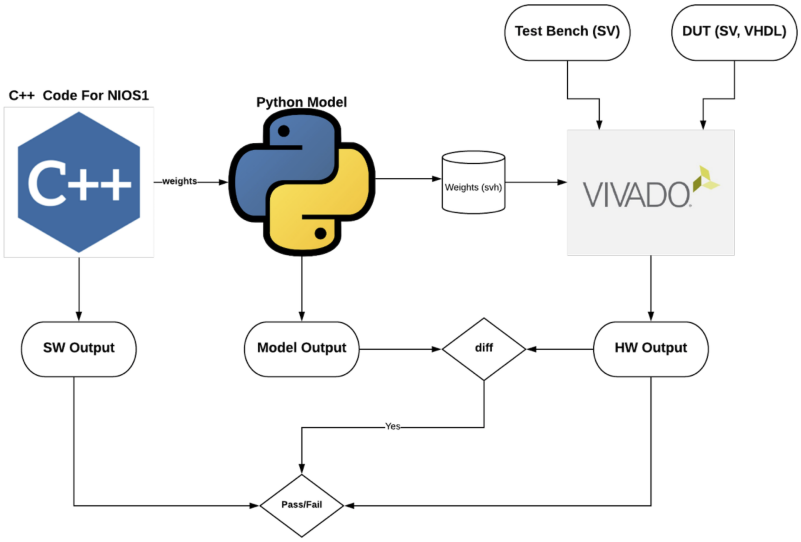

Figure 19: Verification pattern

Our simulation tools allow us to generate outputs through our NeuralCore. This is the way we used to validate our models before implementing them on the board. Here, we can see two different outputs generate with our simulation. It also let us know if the implementation has or hasn't had an impact on our solution.

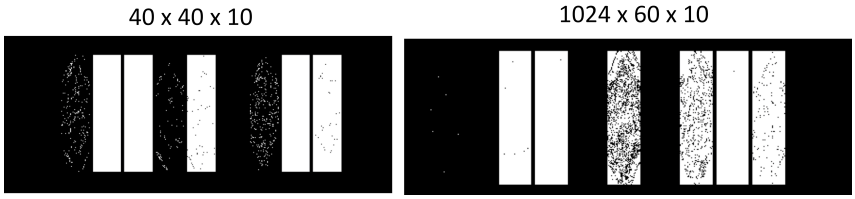

Figure 20: Verification pattern

4 Conclusion

This project is focused on optimizing a neural network by using a SoC to implement a hardware/software acceleration. We started from the software application code to understand how it works. We proceeded with small software improvements and profiling the application. This gave us hints on where to work to optimize the execution.

We lost a lot of time using Quartus. We firstly tried to use block diagrams but it was really hard to debug since Quartus is a bit confused when there is errors in the bdf file. One other big problem of Quartus is that all the documentation is still migrating from Altera to Intel. It's hard to find a similar bug on the Intel forums right now. The project was consequent so the time to compile our solution really stole a lot of time.
The project is published on GitHub under the MIT license which allows people the use this project under conditions. To improve the project, we would suggest to use more recent Quartus tools (at least for the labs) since the old version is extremely buggy (especially Eclipse). Upgrading the PCs could potentially help with compile time which is the big bottleneck of the design process. In this project, we learned to used Altera tools and to sharpen our HDL skills. We explored many IPs and how to use them. We learned how to integrate many processors into a single system. Overall, it was a good experience in terms of understanding the whole top to bottom hierarchy of software/hardware development.

To conclude, hardware modules are essential to optimize an application. The SoCKit platform allows us to use a hybrid solution where all hot nodes are hardware modules. This solution is tailored to this application, but since we haven't forgotten to take care of different sizes of inputs and different neural networks, it is also flexible and dynamic. We managed to execute the longest scenario in under 300 ms which represents a speedup of approximately 14000 times. This demonstrates the power of self-tailored solutions to complex problems.

5 Bibliography

References

[1] Farabet, C., Poulet, C., Han, J. Y., LeCun, Y. (2009, August). CNP: An fpga-based processor for convolutional networks. In Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on (pp. 32-37). IEEE.

[2] Riviello, A., M, Hossein, H, Nathan, ELE8307: Rapport d'étape (2018).