Hardware/Software Co-Design
Simple CNN Report

1 Introduction

In this lab, we aim to optimize a Convolutional Neural Network (CNN) using High-Level Synthesis (`HLS`) on a `Zybo` development board based on the `Zynq-7000` SoC. This type of Neural Network is extremely used today to detect patterns and analyze images.

As shown in Figure 1, the CNN developed in this project takes input images in `.ppm` format, each consisting of three channels — red, green, and blue — represented as 88×88 matrices of unsigned 8-bit pixels. The network classifies each image into one of ten possible categories: Airplane, Bird, Car, Cat, Deer, Dog, Horse, Monkey, Ship, or Truck.

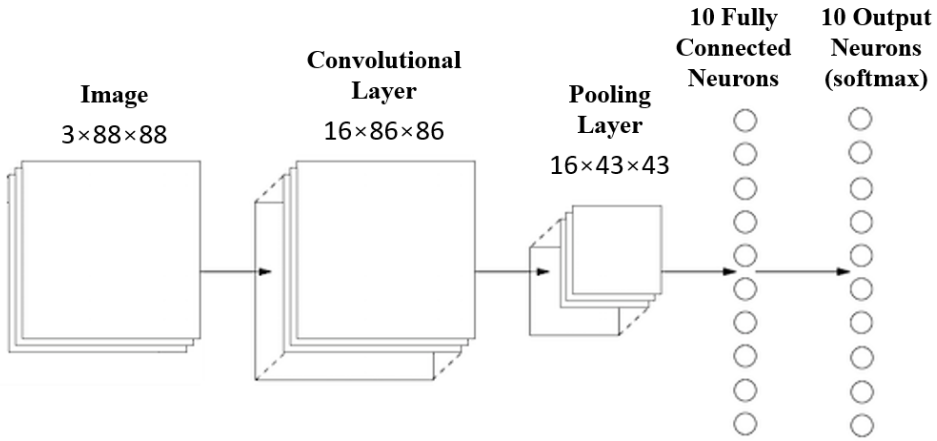

Figure 1: Diagram of the simple Convolution Neural Network to be implemented in this lab assignment.

In the preprocessing stage, the pixel values of the input image are scaled to the interval $[-1, 1]$, as defined in Equation 1. Following this, the image is processed through 16 three-dimensional convolutional filters, each with a kernel size of 3×3×3. These convolutions produce sixteen 86×86 feature maps that capture different aspects of the image. The resulting feature maps are then passed through a ReLU (Rectified Linear Unit) activation function, followed by a pooling layer with a 2×2 window and stride 2. This pooling operation reduces the spatial dimensions, yielding a 16×43×43 output.

$$scaled\ pixel = \frac{\frac{pixel}{255} - 0.5}{0.5} \tag{1}$$

Next, the resulting 3D matrix is flattened into a one-dimensional vector and passed through a fully connected layer. This layer performs a matrix-vector multiplication followed by the addition of a bias vector, producing a 10-element output vector. Finally, a softmax function is applied to this vector, as described in Equation 3, to compute the class probabilities. The class with the highest probability is selected as the predicted label for the input image.

The 3D convolution operation involves sliding a 3D kernel across the input volume, which encompasses all three color channels of the image. At each position, the kernel and the corresponding subvolume of the input are element-wise multiplied and summed, generating a single scalar value. This value is stored in the corresponding location of the output feature map. An example of this process is illustrated in Figure 2.

The ReLU activation is an element-wise operation defined by Equation 2, which outputs the maximum between each input value and zero.

$$y = max(x, 0) \tag{2}$$

The pooling layer employs a max pooling operation with a 2×2 kernel and stride 2, selecting the highest value within each non-overlapping 2×2 region of every feature map, as shown in Figure 3.

The fully connected layer is composed by a matrix-vector multiplication and a vector addition. For the matrix-vector multiplication, the 3D input matrix is flattened in only one dimension and then multiplied by a matrix of weights. Then, the resulted vector is added with the bias vector.

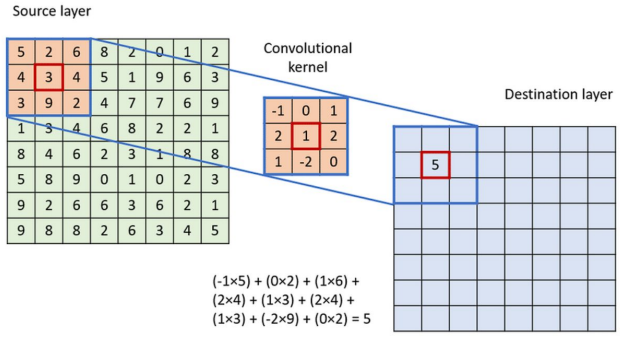

Figure 2: Example of the Convolution algorithm.

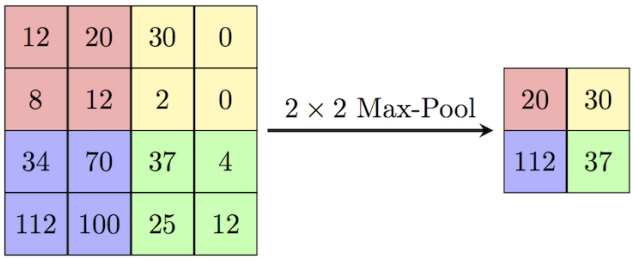

Figure 3: Example of Max-pooling.

The final softmax layer converts this vector into a probability distribution over the classes by applying the softmax function described in equation 3. The class with the highest probability is selected as the final prediction of the model.

$$y_i = \frac{\exp x_i}{\sum_{j=0}^{n} \exp x_j} \tag{3}$$

2 Base Application (SW)

The base application is implemented as a set of C functions, each corresponding to a specific layer of the neural network and operating on floating-point data. A simple for loop iterates over the input images, generating predictions for each one. Pixel scaling is handled by the `normalize_image` function. The CNN is modularized into the functions `forward_convolutional_layer`, `forward_maxpool_layer`, `forward_connected_layer` and `forward_softmax_layer` that applies the 3D Convolution, Max Pool, Fully Connected, Softmax layers, respectivly. The `forward_convolutional_layer` function also applies the ReLU layer.

3 Optimizations Overview

3.1 Fixed-Point Arithmetic

An initial optimization to improve the throughput was to use fixed-point arithmetic in the convolution, max-pooling, and fully connected layers, replacing the more computationally expensive floating-point operations. Specifically, all weights were initially converted to the `Q1.15` format, and each pixel value was also converted to `Q1.15` after normalization.

During a convolution operation, 27 multiplications of `Q1.15` numbers and 27 additions are performed, resulting in a `Q6.30`, which is truncated and stored in a `Q6.26` format to ensure reasonable precision without requiting more than 32 bits. The outputs of the fully connected layer are represented in `Q21.41` and extended to `Q23.41` for storage in 64-bit signed integer variables. Before applying the softmax function, these fixed-point values are converted back to floating-point to compute the final class probabilities.

3.2 Parallelizing the 3 Channels

As done in the previous laboratory, we opted to process the 3 channels of the image at the same time, now even more effective due to the convolution being tri-dimensional, ideally, this cuts the time to process a single image by 3 fold.

3.3 Parallelizing the Max-pooling

On top of the previous optimization, we parallelized on the convolution level in order to have a new max-pool value at each nine cycles of the inner loop. Meaning, we are now doing 4 convolution output cells at the same time. Since each cell takes 9 kernel passes to be computed, at the end of the 9 passes we can do the max-pooling method get a new output values. A visualization of this algorithm can be seen in Fig. 4.

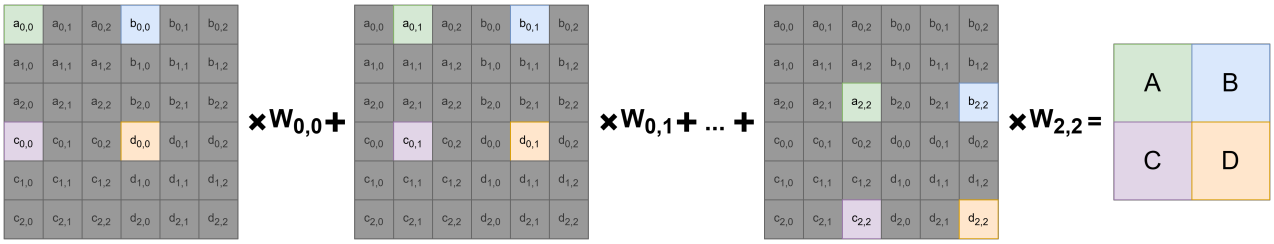

Figure 4: Example of the parallelization at the convolution level for 4 output values and a single kernel.

What this now means is that our convolution kernel is no longer just convolution, but also max-pooling. If we have an $3 \times 88 \times 88$ input image, it's output is a 43×43 array ($\times$OFM). While we are at it, the `ReLU` is also done at this stage.

Since at the end of the IP, we had enough resources, an additional layer of optimization was implemented and now the IP does 8 output cells at the same time, resulting in 2 max-pool results each 9 cycles of the inner loop. A visualization of this is similar to what is in Fig. 4 but with 4 more (2×2) output cells to the right. However, the output width after pooling is 43, which is an odd number. When performing two max-pooling operations at a time, the IP processes an even number of elements per row (44), resulting in one extra value per row. To simplify the IP design, this discrepancy is handled in software: additional zero weights are used in the fully connected layer to multiply these extra max-pooled elements, effectively nullifying their influence on the final 10-element output vector.

3.4 Direct Memory Access - DMA

To address the overhead introduced by AXI-Lite during matrix and weights transfers, the IP was reconfigured to use AXI-Stream. A DMA (Direct Memory Access) controller was then integrated to manage stream-to-memory-mapped communication via High-Performance (HP) Ports, replacing the less efficient General-Purpose (GP) Ports used by AXI-Lite. This process removes the handshakes made by each transfer to the IP and allows us to send the data of the matrices in bursts of words. To interact with the DMA, a simple pooling was used and the interrupts are disabled.

3.5 Sending Weights Once

Since each convolution operation uses the same set of weights, the weights are transferred to the IP only once via DMA. No additional AXI-Stream control logic was implemented to distinguish between weight and image transmissions. Instead, the process was simplified by assuming that the first DMA transfer always contains the weights, while all subsequent transfers correspond to input images for processing.

3.6 SW-HW Parallelism

After integrating the DMA, the new bottleneck of the system became the convolution operation performed by the IP core. This is because the DMA transfer time and the time required by the CPU to perform image normalization, as well as the fully connected and softmax layers, is shorter than the time taken by the IP to execute the 3D convolution. To reduce the overall execution time, CPU processing (normalization, fully connected, and softmax layers) was overlapped with the IP convolution through parallel execution.

The pipeline operates as follows: initially, the CPU normalizes the first image and sends it to the IP core. Once the transmission is complete, the CPU immediately begins normalizing the next image. When the IP finishes processing the first convolution, the CPU sends the next normalized image and, after completing the transfer, proceeds to process the fully connected and softmax layers of the previously convolved image and normalize the next image. Then, the CPU continues to do the fully connected, softmax layers and normalization while the IP does the convolution until all images have been processed. This parallelization strategy hides the CPU processing time within the IP convolution time.

Since both the image normalization and the fully connected layer require significant memory bandwidth, so to minimize contention on the memory bus between the CPU and DMA, the CPU waits for the DMA to finish reading the image data before beginning its own operations.

Another important detail is that the CPU reads the results of the previous convolution while the IP is generating the output for the next image. To avoid concurrency issues, a double-buffering mechanism is used: at each iteration, the IP writes its output to one buffer while the CPU reads from the other buffer — written by the IP in the previous iteration.

4 Application HW/SW

4.1 HW IP

4.1.1 Specification HLS

The IP has simply two communication channels, the input stream and the output stream, both with 64bit width, meaning there is no control with the `AXI-Lite` protocol. The first time the IP is ran, it expects to receive the bias values followed by the kernel values, since each of this values comes in `Q1.15` (16 bits) format we receive 4 of them at a time and store them in their designated BRAMs. This values will be common through out the entire execution, therefore this process will only occur once.

We then enter the main body of the IP, where we will first expect the input stream to give us the 3 channels of an image (first the Red, then Green and then the Blue) to store these values on their designated BRAMs. Same logic for the kernel and bias values, since the information about the pixels comes in `Q1.15` we receive 4 at a time.

After we have the information of an image we start by performing a similar operation to the one done in the previous laboratory convolution, described in Fig. 2, only this time we will apply different `OFMs` to the same image. Like mentioned before, we will calculate the result value of the convolution of 8 output cells at the same time - Fig. 4 - once we are done, we will be left with 3 (channels) accumulator values for each of the 8 outputs. Since both the kernels and image pixels have 16 bits, the accumulator required 31+5 bits to accommodate the result of the multiplication and the accumulation of 27 sums. Figure 5 displays the hardware for this operation.

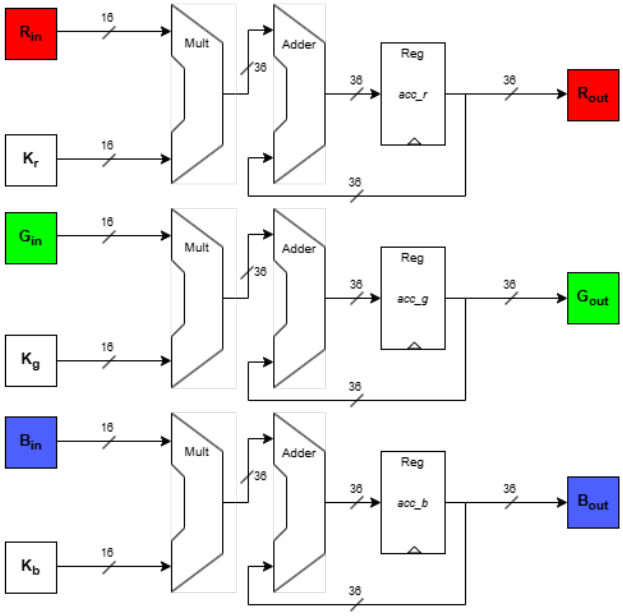

Figure 5: RTL of the convolution hardware for a pixel of an image.

We then sum, per output, the values of the channels with the bias, do the max-pooling and truncate the values to 32 bits to then apply the `ReLU`, now it's only a matter of streaming out this 2 32 bit values in a single 64 bit word. Since this process is being pipe-lined, when we are streaming out this values we are already performing the convolution of the next ones. The figure bellow - Fig. 6 - displays the described IP. The `Conv Cells` are a representation of the hardware in Fig. 5, where we calculate the convolution of a single output value.

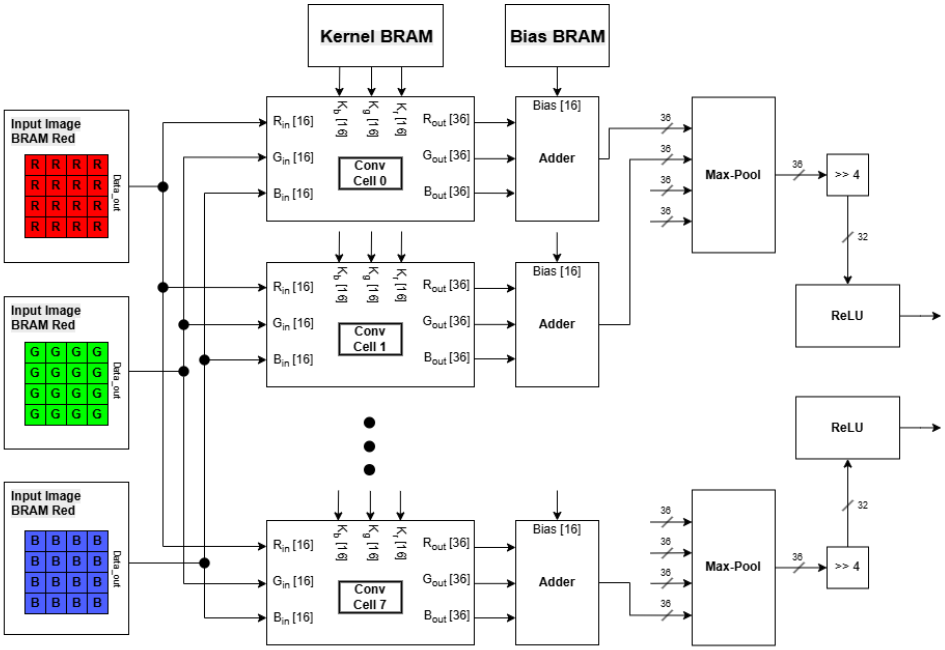

Figure 6: Full diagram of the IP, disregarding control and streaming logic.

4.1.2 C Simulation

Initially, a simple C-based simulation was used to verify the correctness of the HLS-generated IP. The testbench first sends the convolution weights to the IP via an AXI-Stream interface. Then, in each iteration of the main loop, it transmits a scaled image in fixed-point format and reads back the resulting output through the stream.

After receiving the output, the testbench computes the convolution and max-pooling layers in both fixed-point and floating-point arithmetic on the CPU. This allows verification of correctness by comparing the IP output with the fixed-point software implementation (for exact match) and with the floating-point version (to ensure numerical closeness, using a tolerance of 1E-3).

Additionally, the testbench processes two images to confirm the expected behavior of the IP: it should treat the first transmission as the weights and all subsequent ones as image data.

4.1.3 C/RTL Co-Simulation

Figure 7 shows the C/RTL Co-Simulation waveform for the HLS IP for two images with `IMAGE_HEIGHT=88`. In Figure 7 is possible to verify that in the real cases (`IMAGE_HEIGHT=88`) the total execution time is dominated by the convolution computation, not by receiving the matrices in the input stream, since only a small portion of the time is spent in the `stream_in` communication. Additionally, Figure 8 indicate to us that the IP spends 250 ns (25 cycles) to compute two max-pooling values in parallel, resulting in a throughput of 1 max-pooling element/125 ns.

4.2 Architecture HW/SW

4.2.1 DMA

The DMA was integrated to the system by using the Vivado IP Catalog. The DMA was responsible to act as bridge between the AXI-Stream ports of the IP and the AXI-Full High Performance Ports of the PS, which access the DRAM. The DMA was configured to operate with the maximum burst length available (256). In addition, the buffer length was increased to $2^{17} - 1$, since the output of the max-pooling has size equal to $16 * 44 * 43 * sizeof(int) = 121088\ bytes$. Also, the stream and memory-mapped ports of both read and write channels of the DMA are configured to 64 bits, them accommodating 4 16-bit inputs (weights or pixels) and 2 32 bit outputs (max-pooling result) in the bus.

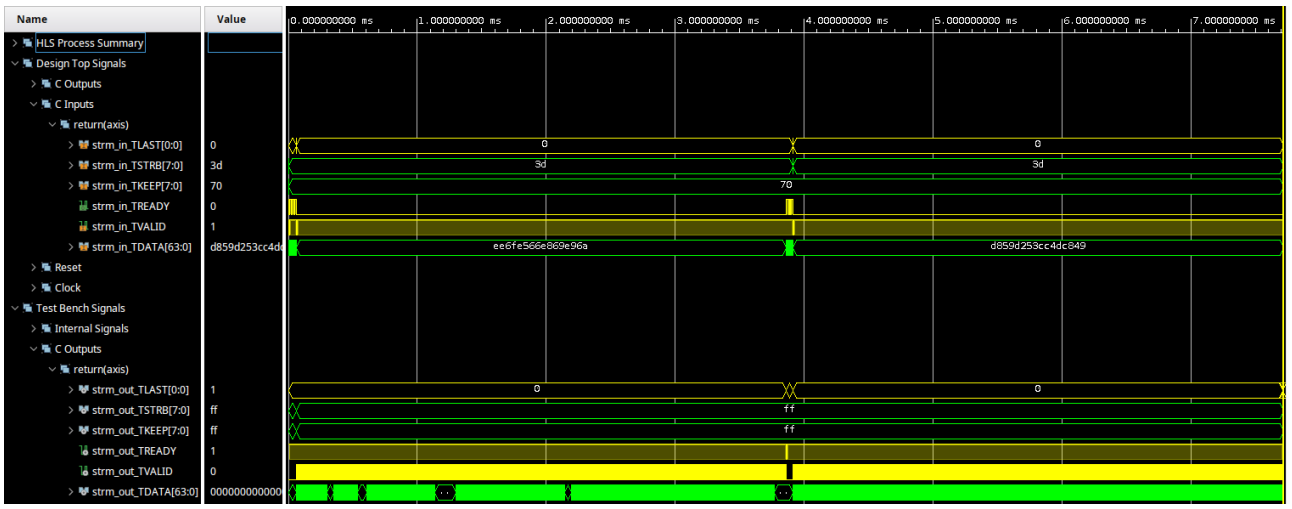

Figure 7: C/RTL Co-Simulation for HLS IP for two images with `IMAGE_HEIGHT=88`

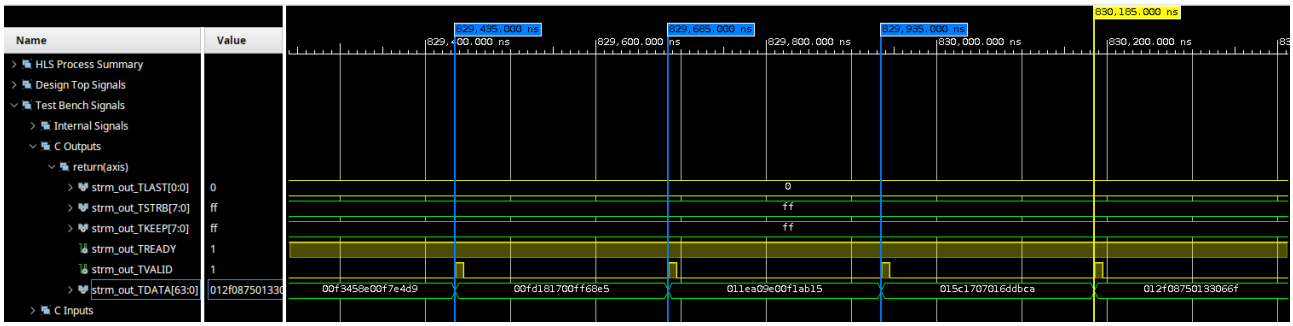

Figure 8: Generation of max-pooling values in C/RTL Co-Simulation for `IMAGE_HEIGHT=88`

4.2.2 Software

First, the application code calls the `init_sw_pipeline()` function, which is responsible for initializing the pipeline. This function disables DMA interrupts, transfers the convolution weights to the IP via DMA, computes the first normalized image, and configures the DMA to send this image to the IP and to receive the result of the convolution. The function returns after the DMA completes the transmission of the first image.

After the main loop starts, which calls the `forward_convolutional_layer_hw` function. This function normalizes the next input image, waits for the IP to complete the current convolution, and configures the DMA to send the new image to the IP and receive the result into the alternate buffer. As previously explained, a double-buffering scheme is used to allow the CPU to read the output of the previous convolution while the IP processes the next image. The function then waits for the DMA to finish transmitting the current image.

Afterward, the functions `forward_connected_layer_int` and `forward_softmax_layer_int` are called. These functions are analogous to the functions `forward_connected_layer` and `forward_softmax_layer` but for fixed-point arithmetic.

Additionally, the steps of normalization and fully-connected layer are further improved to accelerate the software. In normalization, the division by $0.5F$ was distributed and the division by $\frac{1}{255*0.5F}$ was approximated to a multiplication by $0.007843138F$. Also the scaling and fixed-point casting operations were fused into a single loop to reduce loop overhead.

In the fully connected layer, the matrix multiplication was simplified to a matrix-vector multiplication, with the accumulator initialized using the bias values — fusing the matrix-vector multiplication and bias addition loops into one. Also, the loop ranges are modified to use constant values instead of run-time variables to simplify the code. All these operations are performed in fixed-point arithmetic to further accelerate computation.

4.2.3 Integrated Logic Analysis (ILA)

The Integrated Logic Analyzer (ILA) was incorporated into the design as a runtime debugging tool to probe AXI signals and monitor hardware behavior. It was configured to observe all AXI-Stream signals as well as the control signals of the AXI-Full interface. In this case, the ILA was used to identify unexpected behavior in the DMA, which was caused by incorrect handling of the `keep` and `strb` signals by the IP. Specifically, the IP was incorrectly sending `0xF` instead of `0xFF` for both signals, due to the fact that the signal logic was not updated when the IP was adapted from a 32-bit to a 64-bit bus interface.

4.3 Implementation and Results

The System Block Diagram - displayed in Fig. 9 - shows the final architecture of the accelerator, here we emphasize the use of a single DMA block and only one IP. The simplicity of our diagram comes from the fact that we only need to stream in the bias, weights and images and stream out the max-pooling results, where this transfers do not represent the bottleneck of our system, hence there is no need for additional memory interfaces.

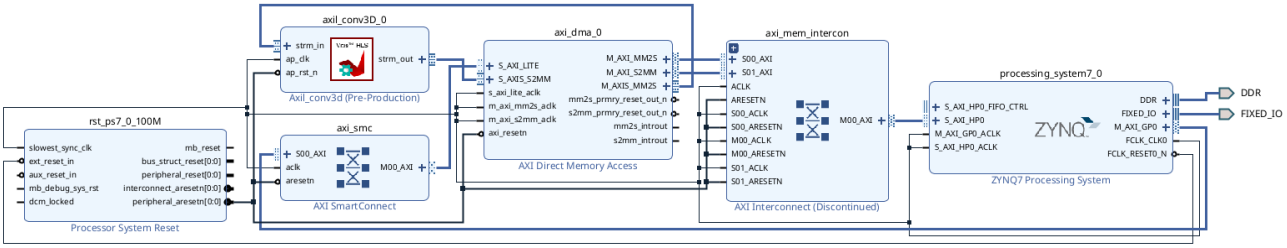

Figure 9: System Block Diagram taken from Vivado after block design.

Table 1 summarizes the results from the synthesis performed in Vitis and the implementation carried out in Vivado. In all cases, Vitis overestimates the resource usage when compared to the actual values reported by Vivado. Additionally, due to loop unrolling applied to the convolution layer to increase parallelism, all available DSPs on the target boards are fully utilized. Vitis estimates the clock period with an uncertainty margin of 2.7 ns; the actual clock was inside of the expected range, indicating that the Vitis estimations are reasonably accurate.

	Ideal Clock (ns)	Power (W)	LUTs	FFs	BRAMs	DSP
Vitis	7.2	-	8774	6814	30	86
Vivado	9.36	1.849	6324	6460	25	80

Table 1: Estimations for the IP generated by each tool

Table 2 show the results of the implementation per component from Vivado. Note that the communication components (DMA + AXI) introduces big resource overhead, corresponding to 40.51% of the LUTs, 55.45% of the FFs and 36% of the BRAMs relative to the total usage. As previously discussed, all available DSPs are utilized in order to maximize the computational power of the IP.

	LUT	FF	BRAM	DSP
DMA	1576	2343	9	0
AXI	986	1239	0	0
Convolution 3D	3747	2845	16	80
Reset System	17	33	0	0

Table 2: Vivado Resource Utilization per IP component

5 Results

Table 3 presents the binary size and execution time for both code versions. The hardware-software have a 0.9 ms initialization overhead, due to the software pipeline, and more 0.967 ms per image. The hardware implementation achieves a 10.11x asymptotic speedup and a 9.30x speed up for ten images over the baseline software version. This significant performance gain results from a combination of key optimizations: exploiting data-level parallelism within the IP, using DMA to minimize memory transfer overhead, applying software pipelining to overlap CPU and IP execution, and optimizing CPU code bottlenecks.

Moreover, the biggest speedup is achieved by accelerating the convolution operation and combining it with max-pooling. This optimization reduces the execution time of these layers from 8.26 ms to 0.967 ms, resulting in an 8.6× speedup. Additionally, due to the parallelism between the CPU and the IP, the total execution time per image is approximately equal to the IP's execution time.

Further optimizations were applied to the fully connected layer, reducing its execution time from 1.25 ms to 0.51 ms, which corresponds to a 2.45× speedup. In the image normalization step, the hardware-software implementation introduces an overhead due to the conversion of floating-point values to `int16_t`. However, various optimizations have minimized this overhead to only 0.01 ms, resulting in a slowdown of just 3.7%. Since the execution time of the softmax layer in the base application is negligible (0.001 ms), no optimizations were applied to it.

Attempting to estimate an execution time for our accelerator, we get that a new result (2 max-pool results) each 3 loop iterations (cycles), since we do a 3-stage pipeline between the 3 lines of the kernel, so ideally at each 3 cycles, 3 multiplications of the last row of one kernel for 8 different output pixels will be done. Since the output max-pooling matrix has dimension $16 \times 44 \times 43$, then the IP spends 45410 cycles doing the operation, as can be see in equation 4.

$$3 \times 44 \times 43 \times 16 \times \frac{1}{2} + 2 = 45410 \text{ cycles} \tag{4}$$

Ideally, the IP can receive four image pixels per cycle and only one additional cycle is needed to send the result matrix to the DMA (since the transmission will be overlapped by the convolution computation), resulting in a estimated total number of 51219 cycles, which results in 0.512 ms, as shown in equation 5.

$$45410 + 88 \times 88 \times 3 \times \frac{1}{4} + 1 = 51219 \text{ cycles} \tag{5}$$

Unfortunately, the actual computational time of the IP is significantly higher than the initial estimates. To obtain a more accurate approximation, the RTL generated by Vivado was analyzed. This analysis revealed that the IP performs the three multiplications for each kernel row in parallel. Assuming the IP requires five additional cycles per kernel row to compute indices, load data from the BRAMs, and write the results, the total number of cycles is calculated as 96,624 or 0.966 ms (see Equation 6). This value is extremely similar to the measured time shown in Table 3 (0.967 ms), indicating a high level of precision in the new estimation.

$$6 \times 44 \times 43 \times 16 \times \frac{1}{2} + 88 \times 88 \times 3 \times \frac{1}{4} + 1 = 96624 \text{ cycles} \tag{6}$$

Despite the integration of hardware acceleration, the binary size of the hardware-software version is only 1.18x larger than that of the original software version — which is acceptable considering the great performance improvements.

	SW-Only	HW-SW
Binary Size (Bytes)	92,656	109,152
Image Normalization Execution Time (ms)	0.27	0.28
Convolution 3D and Maxpool Execution Time (ms)	8.26	0.967
Fully Connected Layer Execution Time (ms)	1.25	0.51
Softmax Layer Execution Time (ms)	0.001	0.001
Execution Time per Image (ms)	9.78	0.967
Total Execution Time for Ten Images (ms)	97.83	10.52

Table 3: Execution Time and Binary Size for each code version with `-O3` flag

About the resource utilization, since each convolution multiplication is between 2 16 bits integer and the IP realizes 72 multiplications in parallel (3 kernel elements times 3 channels times 8 output pixels), so it requires 72 DSPs for the convolution multiplications. Also, for compute the image and kernel base indices, it's necessary to do more 3 32 bit multiplication, which requires more 9 DSPs, totalizing 81 DSPs. The Vitis estimation requires 86 DSPs, probably because some of the sums are implemented as DSPs. However, note that the Vivado only requires 80 DSPs, because the Vivado is capable of checking the maximum size of each index and optimizing the size of the multiplier regarding this information, which can reduce the total number of required DSPs.

Regarding resource utilization, each convolution multiplication operates on two 16-bit integers, and the IP performs 72 multiplications in parallel — calculated as 3 kernel elements × 3 channels × 8 output pixels — resulting in a utilization of 72 DSPs for convolution. In addition, computing the base indices for the image and kernel requires three 32-bit multiplications, which consume 9 additional DSPs, bringing the total to 81 DSPs.

From Table 1, the Vitis estimation reports a requirement of 86 DSPs, probably because some addition operations are also implemented using DSPs. In contrast, Vivado reports a total of only 80 DSPs. This reduction is attributed to Vivado's ability to analyze the maximum possible size of each index (since the loop bounds are constant) and optimize the multipliers accordingly, then reducing the DSP usage.

As for the BRAM estimation, the design includes 3 arrays of `image16_t` (16 bits), each with a size of $88 \times 88 + 2$, meaning each one has a total size of 15492 Bytes or 123936 bits. If a BRAM holds 32kbits, approximately 4 BRAMs are required per image array, totaling 12 BRAMs for the images.

For the weights (864 bytes) and the bias (32 bytes), since they are declared in separate arrays, two additional BRAMs are allocated - over-sizing the dimensions of this variables but allowing parallel loads. This leaves 2 BRAMs away from the Vivado estimation, see Table 1. Analyzing the RTL, reveals that the system split the weights array in 3 (for each channel), allowing multiple reads in parallel, making the 16 BRAMs.

In the case of Vitis, synthesis logs indicate that the bias is not stored in BRAM, reducing to 15 BRAMs. Since in Vitis reports BRAM are 16 Kb blocks, then this corresponds to 30 BRAMs in its reports.

6 Possible Improvements

Even though we have a near-optimal solution, there are still improvements that could be made for a more efficient application when it comes to the IP.

We could start by changing the order of the unroll done in the convolution, meaning instead of doing 8 outputs at a time - taking 9 inner loop cycles to conclude the 8 outputs; we could instead have each iteration calculate the entire convolution of an output cells (9 multiply and accumulate) - taking only 8 inner loop cycles to compute the 8 values. Now we would have 2 max-pool results each 8 loop iterations instead of 9.

Another possible improvement on the IP is better partitioning the `kernel/bias` BRAM's to ensure that all 3 channel kernel values are loaded in parallel, saving clock cyles per iteration to load these values.

Lastly on the application side we could attempt to utilize the second core of the `ARM Cortex-A9`. This would result in an heterogeneous system approach to parallelization where the "main" core would work with the IP accelerator and then a smaller batch of images would be given to the second core to work in parallel in a full software approach. However, we would be bottle-necked by the memory bandwidth and, whilst with more computational power. Additionally, we can try to use `ARM NEON` instructions to accelerate the software operations.

7 Conclusion

This laboratory work successfully optimized a CNN for image classification through hardware/software co-design on the Zynq-7000 SoC platform. The implementation employed fixed-point arithmetic using `Q1.15`, `Q6.26`, and `Q23.41` formats across convolutional, pooling, and fully connected layers, substantially reducing computational overhead while preserving adequate precision. Critical parallelization strategies included simultaneous processing of all three RGB channels and concurrent computation of eight convolution outputs per cycle, enabling efficient generation of two max-pooling results every nine cycles.

DMA integration minimized data transfer, effectively eliminating AXI-Lite handshake overhead. A sophisticated SW-HW parallelism approach with double-buffering allowed overlapping of CPU tasks— image normalization, fully connected layer execution, and softmax computation—with the IP's convolution operations, then hiding CPU processing time within hardware acceleration time.

The optimized implementation achieved a 10.11x asymptotic speedup and 9.3x speedup for ten images compared to the baseline software, reducing per-image execution time from 9.78 ms to 0.96 ms. This performance gain was accomplished with only an 18% binary size increase (from 92,656 to 109,152 bytes), demonstrating efficient hardware integration. Resource utilization analysis revealed full consumption of the available 80 DSPs through loop unrolling in the convolution layer, while communication components (DMA and AXI interfaces) accounted for 40.51% of LUT and 55.45% of flip-flop overhead.