Instituto Superior Técnico

Mestrado em Engenharia Eletrotécnica e de Computadores

Co-Project HW/SW

Lab 1 Questionaire Report

Alexandre Santos, nº 99884 ares.santos@tecnico.ulisboa.pt
Jorge Miguel M. Contente, nº 102143 jorge.contente@tecnico.ulisboa.pt
Nuno Abreu, nº 103416 nuno.g.tribolet.de.abreu@tecnico.ulisboa.pt

2º Semestre 2024/25

Introduction

This assignment follows the footsteps of the previous one as we are asked to use the power of the ZYNQ-7010 board hardware to speedup processing of our programs. In this case, we are tasked to accelerate the processing of convolutions across numerous images. The convolution is a mathematical operation that, in our context, takes a 3x3 matrix as a kernel and shifts along a 88x88 matrix, where the values inside the kernel are multiplied and accumulated, forming a result matrix of 86x86.

This operation is widely used throughout digital image processing and with the boom of artificial intelligence, convolutional neural networks have become a intrinsic tool in AI development, therefore, its no surprise that there is a important need to accelerate this process.

The code provided by the teachers is an IP that will do just that and, in a following section, we will dive into our process while allying the provided IP with the C program. To operate our hardware we will be accompanied by the AXI-Lite interface, composed of 5 different channels, 3 to send (Write Address, Write Data, and Write Response) and two to read (Read Address, Read Data). These allow communication between the master and slave interfaces, in our case, the VITIS program and the ZYNQ-7010 board, respectively. The data will be parted into 3 for every image, referring to the RGB framework of pixel values, sent to the board with a RRRRR..GGGG...BBBB format, i.e, the R(ed) values will be sent first, followed by the G(reen) and then the B(lue).

Software-only implementation

Our software implementation had the foundation of the professor's code, with no modifications. In this context (SW-only), the provided code simply invokes the function `sw_convolution_2D` 3 times, one for each pixel channel (Red, Green and then Blue), as explained in the introduction section. Each invocation will then execute a convolution given a provided 3x3 kernel, which in turn can be customized.

We proceeded to run the Synthesis and verified the results with the original kernel and the identity kernel, which they were correct. The execution time of our application was 12.39ms.

SW/HW implementation

As stated in the introduction section, our provided IP was the main tool to supply hardware acceleration to our simple SW-only implementation. Thus, we ran a Synthesis, C-Simulation and Co-Simulation with the provided test-bench. This resulted in no errors and we proceeded to the ZYNQ-7010 board. To do so we proceeded to send the component to Vivado and generate a block design along side the PS and the necessary *AXI Lite* blocks.

After obtaining the constraints file, we built the platform and the application. To do so we used the skeleton code provided, making the necessary changes relating to data transfer. This changes related to transferring the images' data to the IP, making use of the board's DDR memory and BRAMs. As images are 88x88 matrices, where every 1 Byte pixel is sent one by one, our DDR memory will store each pixel separately, totaling 88x88 lines, 1B each .

These values will then be sent to the BRAMs, via the *AXI Lite* interface, to a table in the IP. The access to the interface is made through the addresses present in the drivers, as well as the base address provided by Vivado's address map. After the IP finishes processing, data is fetch through the interface in a similar manner, and written back to the DDR memory and later back to our PC.

Our SW/HW solution presented a speedup of 1.15x, which is not as much as we'd like to. However, its expected if we assume that in our scenario the communication time of sending the data is non-negligible, thus affecting our performance. Another issue that might be decreasing our performance is the serial nature of the application, as only after a channel is finished processing that the next one will commence. Therefore, parallelizing the channel processing might then be a possible optimization to our implementation.

The resources estimated and used can be seen in the respective tables below:

LUTs	FFs	DPSs
395	342	5

Tab. 1: HLS Estimate

LUTs	FFs	DPSs
418	342	5

Tab. 2: Real resource consumption

Integrated Logic Analysis

To use the Integrated Logic Analysis, we created a new VIVADO environment where we followed the normal workflow: Synthesis, Implementation and Generate Bitstream. Consequentially, it is now possible to use ILA and check the hardware's waveforms for debugging, shown on figure .

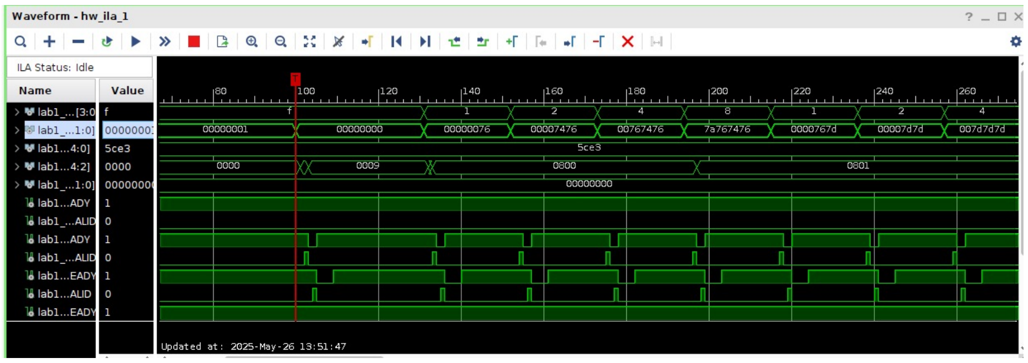

Fig. 1: Integrated Logic Analysis' Waveforms

However, our use of ILA was not extensive, as we opted to use VITIS debugger, which provides very similar debugging capabilities.

Optimizations

Three IPs

The first attempt at optimization was to exploit data level parallelism by implementing three IPs, one for each channel. This would parallelize the computation time, and in theory reduce it by three.

To achieve we added two more IPs in the block diagram and then altered the application code to send each channel to the address of it's respective IP sequentially, wait until all finished, and then receive the output.

This resulted in negligible speedup, from 1.15 to 1.19. This came at 1690 LUTs, an increase of 276% over the original value, not at all worth the speedup.

This result indicates that most of the execution time is taken up by the communication over the AXI Lite interface, which takes upwards of 30 cycles per communication.

Hardware-Software Codesign

We noted that while the IP was doing computation, the PS was idle. To use the complete system we attempted to compute one of the channels on the processor while two other channels were being calculated on the respective IPs.

This was easy to achieve as it only required changing a line of code.

This optimization resulted in a speedup of 1.21. Though it is better then using three IPs, it was still less then expected.

Interleaved Computation

To attempt to hide the data transfer times, we changed the order of operations. After sending the first channel fully, the first IP would start it's computations. This allowed the computations to occur simultaneously with sending the second channel. The same thing was done for the third channel.

The speedup for this implementation was lower then all other implementations. We do not know why this might have happended and suspect possible issues related to the control of the AXI Lite interface.

Optimized Communication

It has become clear to us that the time spent by the program is spent mostly on sending data from the PS to the IP through the AXI Lite interface. It is to note that every element of the image matrix is a single Byte, 8 bits, while the data bus for the AXI Lite interface in 32 bit. For the final implemented optimization we attempted to exploit this fact.

Achieving this required altering the data types referring to the image to be 4 Bytes, i.e. on the application side from unsigned char to unsigned int and on the IP side from *uapint<8>* to *uapimt<32>*, as well as reducing all communication loops by a factor of 4.

After communication, bit-wise operations were employed to ensure the correct calculations were being done.

As previously stated, the communication loops where shortened by a factor of 4. This would lead us to expect a speedup of 4, however the one achieved was 3.00. This is still good and no further optimizations were explored. We expected the additional logic for the bit-wise operations to result in higher resource consumption. This proved not to be the case. In fact we achieved lower resource usage, with the new IP taking up only 366 LUTs, as opposed to the 395 of the original.

Unexplored Optimizations

As previously stated, most of the execution time of the program was spent in communication across the slow *AXI Lite* interface. One possible improvement over this would be to use *AXI Stream.*

While *AXI Lite* taxes upwards of 30 cycles *AXI Stream* takes 1 cycle. However for streaming to work we would need to create a DMA module to connect to the IP. DMA or Direct Memory Access would allow the streaming to be done directly to that memory instead of the BRAM and then accessed by the FPGA.

This should, along with the extra bandwidth and data level parallelism already explored, reach the maximum speedup for this technology.

Conclusion

This lab allowed us to explore the many things to consider when accelerating a program.

Though at first it seemed obvious that faster computation would lead to faster execution, this proved not be the case. It was only after recalling what we explored in the previous lab, the signal of *AXI Lite*, that we figured out communication was the bottle neck. It was once again by using previously acquired knowledge on the *AXI Lite* interface that we were able to shorten those communication times by increasing the bandwidth of communication.

We are now more familiar with the tools we can use when designing hardware and interfacing it it application running on a PS. We have also further developed our though process and are more capable of tackleing the next lab asignment.