Image Processing Optimization Using Pynq FPGA
Project Report

Northeastern University
EECE 4632
Prof. Miriam Leeser

Author: Juan Pablo Bernal
4/9/2021

Abstract

Computer vision has been evolving rapidly in the last two decades. It is being used in a wide range of applications including sensing and navigation, object recognition and tracking, security, and much more. All these require basic image processing to improve the quality of your input data and get better results. The main problem is that image processing involves a lot of repetitive tasks that can get very computationally expensive. When working with software, the complex architecture of a CPU can negatively affect the efficiency of the algorithms. Therefore, in this project, we are going to explore different approaches on an FPGA to improve the processing time of different computer vision techniques.

Overview

This project is focused on optimizing spatial filtering using the Pynq FPGA. The first step in image processing is normally noise reduction to increase the quality of the data. This can be achieved by convolving an image with a smoothing spatial filter. Many different filters can achieve this goal, however, in this project, we only explored averaging and median filters. An averaging filter works by replacing the value of each pixel with the average of the pixels in a window around it. A median filter works similarly but instead of the average, the pixel is replaced with the median value of the window around it. the size of the filter dictates how many pixels will be used to compute the new values. To explore the effect the size of the filter has on the performance, two averaging filters and two median filters were used; each of size 7 and 21 respectively.

Source Code

The code used in this project is divided into two parts, the python scripts that are run in the PS and the C++ scripts that were used to generate the hardware design. The first Python script uses OpenCV to read an image and apply the spatial filters. This gives us a reference point to know how long each process takes to run in the ARM processor and how the output should look like. Moreover, this script generates various txt files containing the input data used by the hardware and the golden outputs that were used to verify the hardware results. The input data was generated using a Python script. It first reads and decodes a jpeg image using OpenCV. Then the borders of the image were replicated around the original image to reduce the complexity of the C++ algorithm. Finally, it converts the image to grayscale and saves the array of pixel values to a .txt file. Moreover, it saved the golden output data after the PS had processed the images with each filter in a similar txt file.

The second part is the C++ functions used to generate the hardware design. Since the FPGA does not allow dynamic data structures, each filter had to be implemented individually with the size hardcoded. The image is loaded from the txt and sent to each function as an array. Then, the array is transformed into a matrix and each filter is applied, creating a window around each pixel, and computing the values by taking the average of the window or finding the median value. Finally, the functions return an array containing the output images without the replicated borders.

Image.1 Input image

HLS Optimization

Each filter was made into a separate HLS project to see how different optimization techniques would affect the different filters. Figure 1 and figure 2 show the Pareto optimal curve of different designs for the averaging filters.

1. Default: this design does not have any kind of optimization and uses short datatype (16bits).
2. Partition: this design completely partitions the arrays used to store the image and the result.
3. Pipeline_out_window: this design pipelines the two loops that generate the window around each pixel and the loops that take the result in matrix form and converts it to array from.
4. Pipeline_out: this design pipelines the loops that take the result in matrix form and converts it to array from.
5. Pipeline_out_imgcol: this design pipelines the loops that take the result in matrix form and converts it to array from and the loop in charge of going through the columns of the image.
6. Unroll_window: this design fully unrolls the two loops that generate the window around each pixel.
7. Unroll_out: this design fully unrolls the two loops that transforms the result matrix into an array to be sent to the DMA.
8. Unroll_out_window: this design combines designs 6 and 7.

More designs were explored, like full pipelining, different data types, loop flattening and fully unrolling, and more. However, these designs were either too complex that the result would not fit on the FPGA or would not modify the Default design at all. Since the main goal is to speed up the image processing algorithm the best design for both filters was Pipeline_out_imgcol.

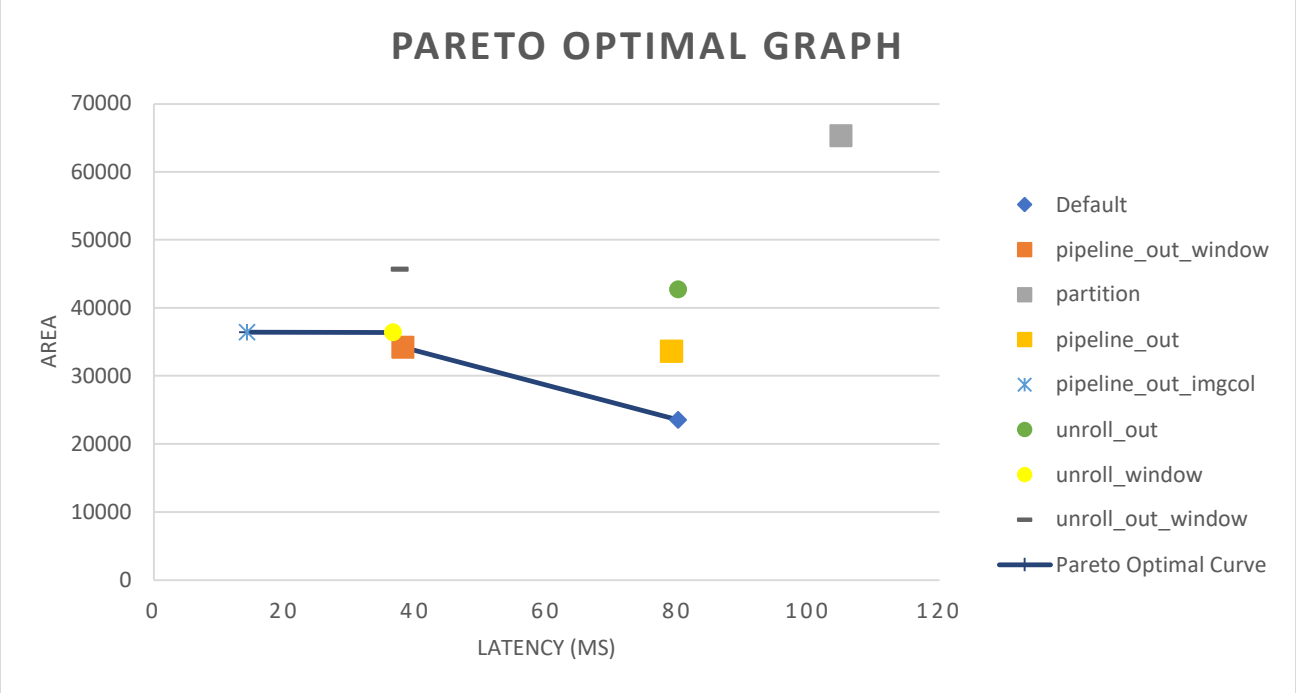

Figure.1 Pareto Optimal Curve Averaging Filter Size 7

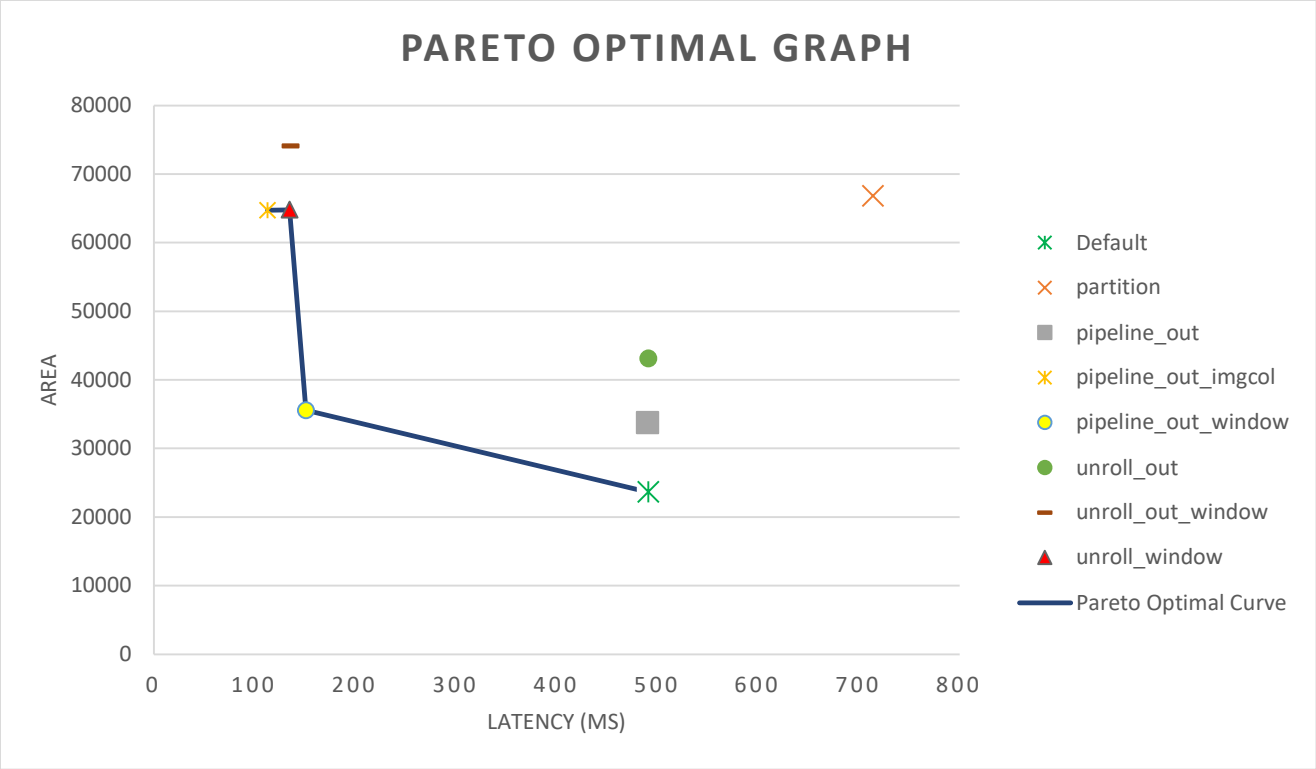

Figure.2 Pareto Optimal Curve Averaging Filter Size 21

Moreover, figures 3 to 4 show the Pareto optimal graphs for the median filters. Since the implementation of this filter is different from the averaging filter different optimizations were explored.

1. Default: this design does not have any kind of optimization and uses short datatype (16bits).
2. Pipeline_out_sort_window: this design pipelines the loops that take the result in matrix form and converts it to array from, the loops in charge of sorting the window array, and the loops in charge of generating the window around each pixel.
3. Pipeline_out_imgcol: this design pipelines the loops that take the result in matrix form and converts it to array from and the loop in charge of going through the columns of the image.
4. Pipeleinie_out_sort: this design pipelines the loops that take the result in matrix form and converts it to array from and the loops in charge of sorting the window array.
5. Unroll_out_sort: this design fully unrolls the loops that take the result in matrix form and converts it to array from and the loops in charge of sorting the window array.

Once more, other designs were explored, however, given that the median filter uses more loops and more complex algorithms, the results would not fit on the FPGA or would negatively affect the Default design. Since the main goal is to speed up the image processing algorithm the best design for both filters was pipeline_out_sort_window

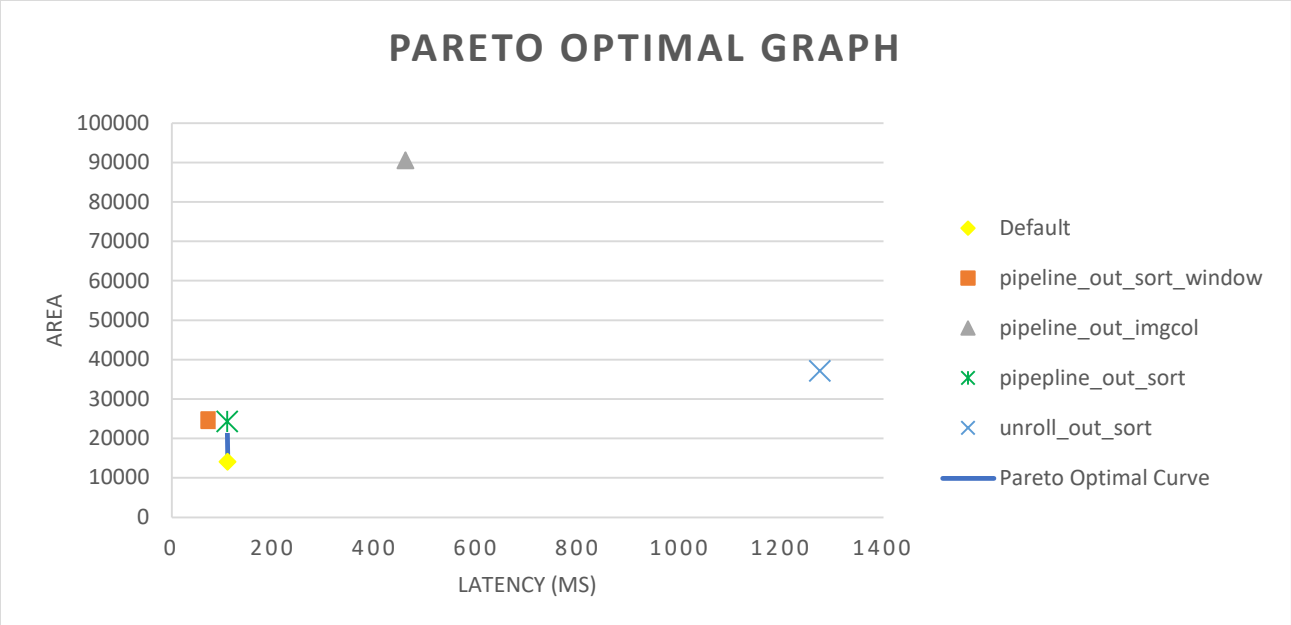

Figure.3 Pareto Optimal Curve Median Filter Size 7

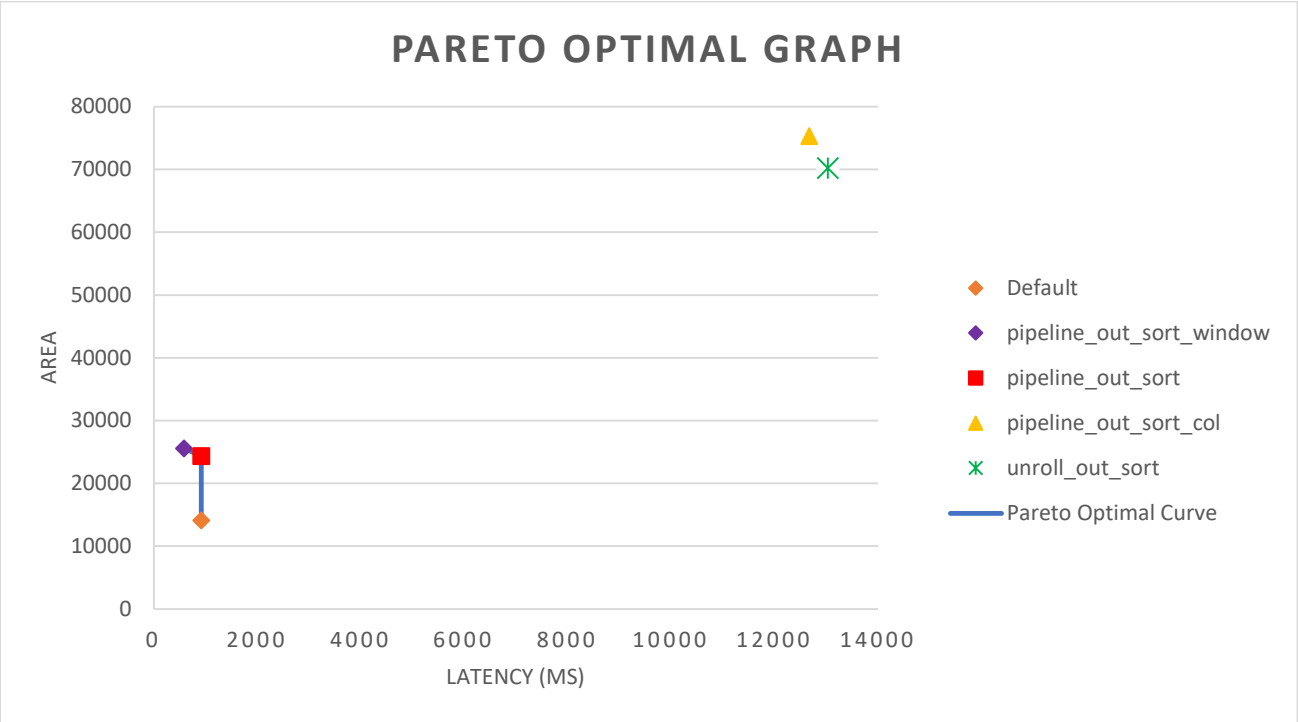

Figure.4 Pareto Optimal Curve Median Filter Size 21

Unfortunately, the optimization was not enough to achieve a faster design in three out of four filters. Since the reference point for the runtimes was calculated with OpenCV functions, which have been greatly optimized, it was difficult to speed up the process with HLS optimization alone. Moreover, these filters use very big loops and must go through every single pixel in the image, the number of computations were too great to achieve the goal. The HLS code was designed with correctness in mind and there are a lot of computations that could have been avoided if better algorithms were used. On the other hand, not everything was a failure, the first averaging filter was optimized enough to speed up the process by a factor of approximately 3.

Vivado Block Design

Once the best design for each filter was selected the IP was generated and loaded to Vivado to design a block diagram. Each filter was designed similarly. The input and output used the AXIS interface to send and receive the data from the PS to the PL. Given that the filters had to compute the entire image before giving the result when only one DMA controller was used there were some problems when trying to only read or only write. Therefore, two AXI DMA controllers were used; one for reading and one for writing. These controllers were connected to the processing system block using AXI interconnect Blocks. Finally, the clock in the processing system was modified to 125MHz (top speed in the Pinq board) to achieve a better time. However, only the averaging filter of size 21 used a clock speed of 100MHz. For some reason, the 125Mhz clock would give the wrong output. Figure.5 shows the block design used for the averaging filter of size 21. All other designs were connected similarly. Finally, the bitstream was generated and copied into the Pinq overlay folder for testing.

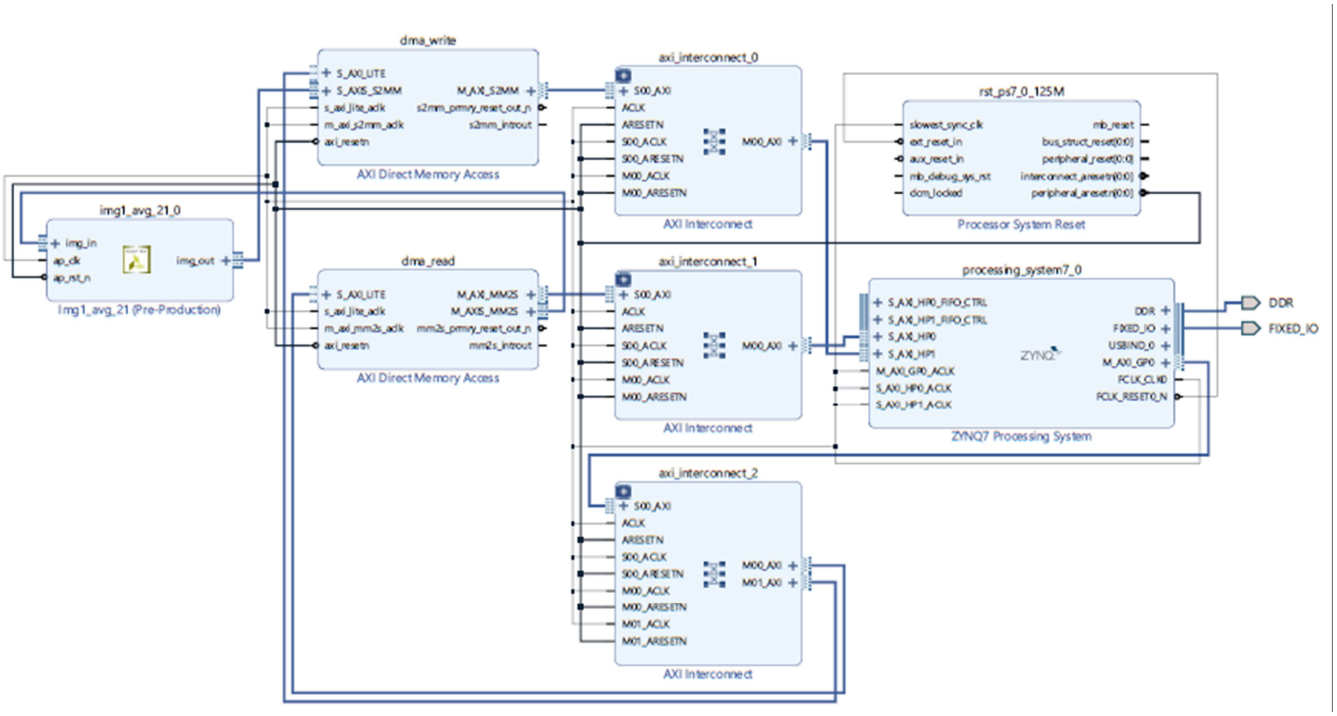

Figure.5 Averaging filter size 21

Validation Results

Finally, with the overlays generated and copied onto the board, the Python scripts for validation were run. In these scripts, each overlay is loaded into the board and the image with the replicated borders is sent to the read DMA controller using a buffer. Then, the time it took the board to run the algorithms was timed. Once the board was done the output was read from the write DMA controller and converted into a numPy matrix. Finally, the txt file containing the golden output (which was previously generated) was loaded and each pixel of the hardware result was compared to it.

Each script was run, and each filter was successful, all four filters matched their corresponding golden output. However, the timing was not great as expected from the HLS predictions. Table 1 shows the timing results. As predicted the only one that experienced a good speed up was the averaging filter of size 7. Which was sped up by a factor of 2.6, all other filters had a worst runtime of that from the PS.

Filter	PS Runtime (s)	PL Runtime(s)	Ratio (PS/PL)
Averaging 7	0.0425	0.016	2.65625
Averaging 21	0.0397	0.115	0.34521739
Median 7	0.0284	1.016	0.02795276
Median21	0.0305	79.06	0.00038578

Table.1 Runtime Results

Conclusion

In retrospect, it can be said that this project was a success. We were able to generate an overlay that accurately implemented smoothing filters in an image, and everything was connected correctly in the FPGA programable logic. Even though this project did not completely achieve its main goal, it was a great learning opportunity. The biggest lesson learned was how different coding hardware is compared to software coding. The mindset needed is completely different since you must be thinking not only about the result you want but also how everything interacts in the physical hardware of the devised used. Moreover, it was a great opportunity to learn about different interfaces and what is needed to successfully program the physical logic inside an FPGA.

In the future, if this project is to be improved, the first thing that needs to be done is to implement more efficient image processing algorithms. There are many ways that a lot of computations can be avoided with better techniques. Also, for the median filters, there are more efficient ways to sort arrays that would save a lot of time. Besides, some features can also be added to the project. It would be good to increase the flexibility of the code. For example, it could accept input images of different sizes or be able to modify the size of the filters depending on the application. Finally, different I/O interfaces can be added to directly send the images to the hardware to avoid the software component altogether.

Appendix A

Source Code Running Instructions:

The code for this project can be found in the GitHub repository https://github.com/juanpbm/FPGA_Image_processing . The image processing Jupiter notebooks can be run without any special instructions. To run the C++ test file each filter must be run individually. For this uncomment the desired filter and comment the others in lines 46 to 53. Each filter is composed of two lines the filter function and the golden output file. Remember to have the txt files in the same folder as the C++ script. Make sure that both lines are uncommented for the desired filter. Moreover, only add the header file for the desired filter in lines 3 to 6. This test file will compare the golden output to the function output. If there is a single wrong value, it will print "Failed at <index of the first wrong value>" otherwise it will print "PASS: The output matches the golden output". To run the hardware, place the folder overlays in the Pyinq board. Similarly, if you want to generate the bitstream or the block design in Vivado use the tcl and hwh files found in the overlays folder.