Final Project Report

Gaussian Blur on the PYNQ-Z2

Ajinkya Joshi

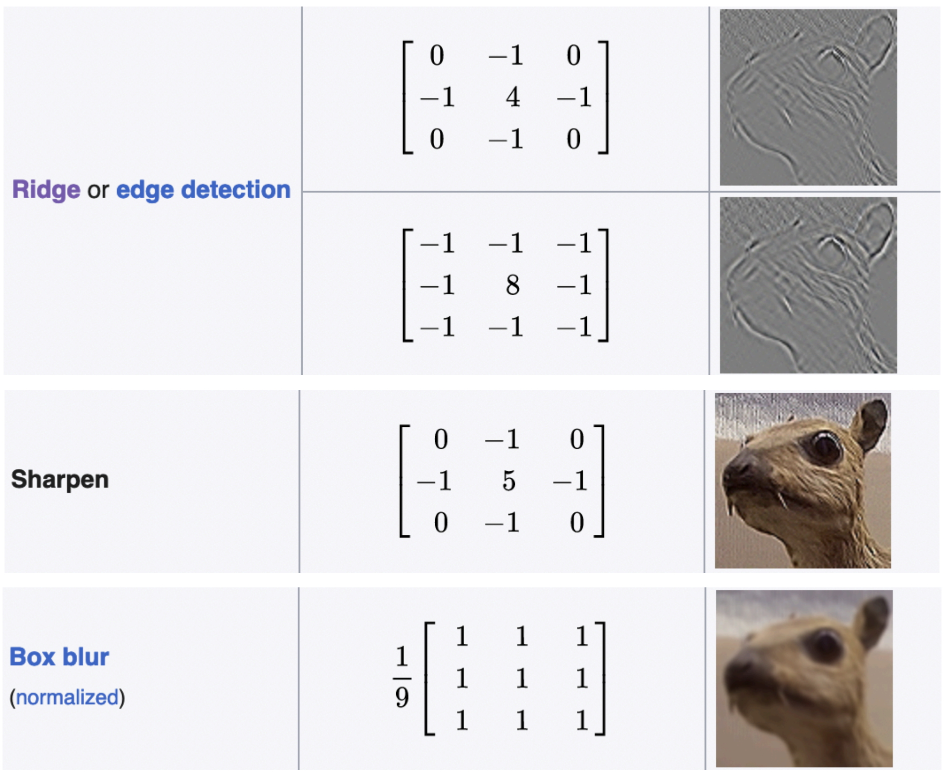
Project Description

Background

The Gaussian Blur executed on the PYNQ-Z2 board is an image processing project that involves blurring an input image using the computing power of hardware and the data processing of software.

Most image processing filters function through a two dimensional convolution algorithm. The input is typically a matrix that has set values defined for each element which represents a single pixel. For colour images, each element in a matrix is defined by an array of three values - a red, green and blue integer. For grayscale images, each element is a single integer that essentially determines the brightness of an image. Each element, for rgb or grayscale, is a value between 0 and 255. For grayscale images, 0 would be black and 255 would be white. Any value in between is a shade of grey.

Another component of image processing filters is the kernel. The kernel is an odd length square matrix whose values determine the final filtered effect at the output. The images below display the different kernel matrices and their effect on the output.



Ridge or edge detection	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

These kernels iterate through each element of the input matrix and find the area that it encompasses. This project refers to this area as the window. The is a matrix with the same dimensions as the kernel but it stores the pixel values that are under the kernel. The window and the kernel go through element wise multiplication and the products are added up together to form a final pixel value. This final value is then inserted in the same iterated element of an output matrix that has the same dimensions as the input.

This project will focus on the gaussian blur which uses the same program logic as any image processing filter. There is a difference between a gaussian and a box blur. As shown above, each element in the box blur kernel has the same values. In contrast, a gaussian kernel looks something like this:

Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{bmatrix}$	

As shown above, the gaussian kernel's centre pixel has the highest magnitude and so will have the greatest influence on the final output of the pixel. Therefore, more information on the original pixel will be retained. The gaussian blur allows one to retain most of the information of the original pixel. This has the effect of making the image clearer than a box blur while simultaneously denoising an image. The kernel can also be of varying sizes, but the dimensions must be odd numbered as the kernel must have a clear centre pixel.

Aim

The aim of this project is to denoise a grayscale image of varying sizes using the gaussian blur filter executed on hardware using a 5x5 gaussian blur kernel. By doing so, I hope to achieve a relative hardware speedup in the time it takes to output a blurred image from the FPGA. This project allows me to understand how best to leverage hardware to the advantage of speedup and power efficiency if this hardware design was converted to a digital IC or ASIC.

Implementation Report

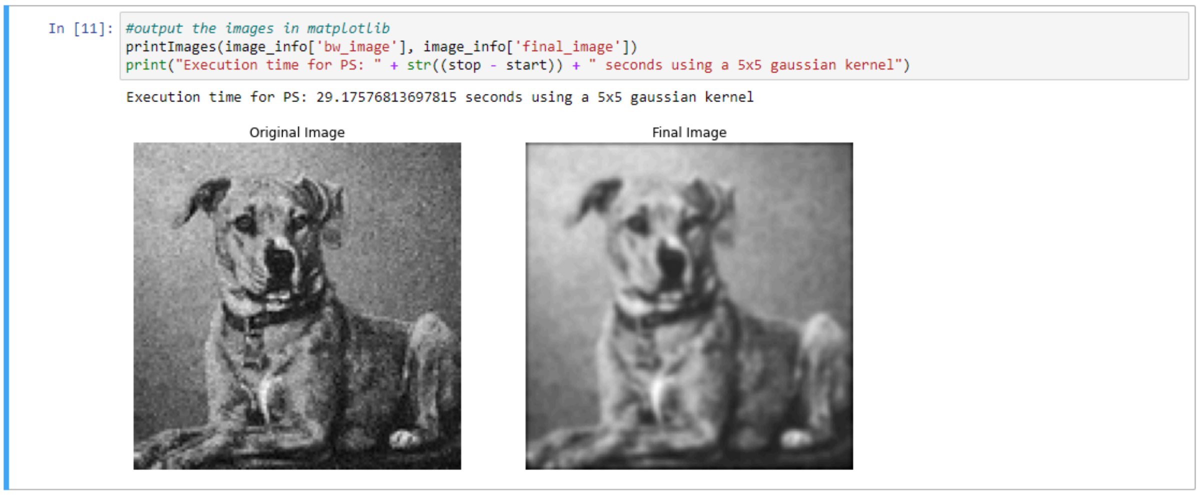
Software Implementation

Before going onto High level synthesis, the filter was first executed exclusively on the ARM cortex A9 processor (on the ZYNQ-7000 SoC) to make sure that the logic worked as expected.

The first step in the software implementation is to gather the inputs. This is done through the OpenCV (CV2) library that is available on the PYNQ board. The advantage of this library is that it views each image as a matrix of pixels. There is also a dedicated function that converts any colour image into a grayscale one. This library is used along with numpy which makes it easier to manipulate large matrices and arrays. Separate functions are created to extract and build images through numpy and CV2.

The matrix is then sent to the gaussian blur function that operates on the input matrix and returns an output matrix with the same dimensions as the input. This function is converted to HLS in the hardware implementation. The function is timed using the time library. The outputs are printed alongside the input, to compare the original and final image, using the matplotlib library.

To test the software, the image ‘doggo.jpg’ is used. This is a 128x128 pixel image. The output of the execution is displayed below.



```
In [11]: #output the images in matplotlib
         printImages(image_info['bw_image'], image_info['final_image'])
         print("Execution time for PS: " + str((stop - start)) + " seconds using a 5x5 gaussian kernel")

         Execution time for PS: 29.17576813697815 seconds using a 5x5 gaussian kernel
```

The execution time for software was approximately 29.2 seconds. The ARM processor is running at a clock speed of 650MHz. This is 6.5x higher than the ZYNQ-7000 FPGA’s 100MHz. Therefore a relative speedup in the hardware execution would be computing a blur at a time less than 189.6 seconds. Any execution time below that would be considered a relative speedup.

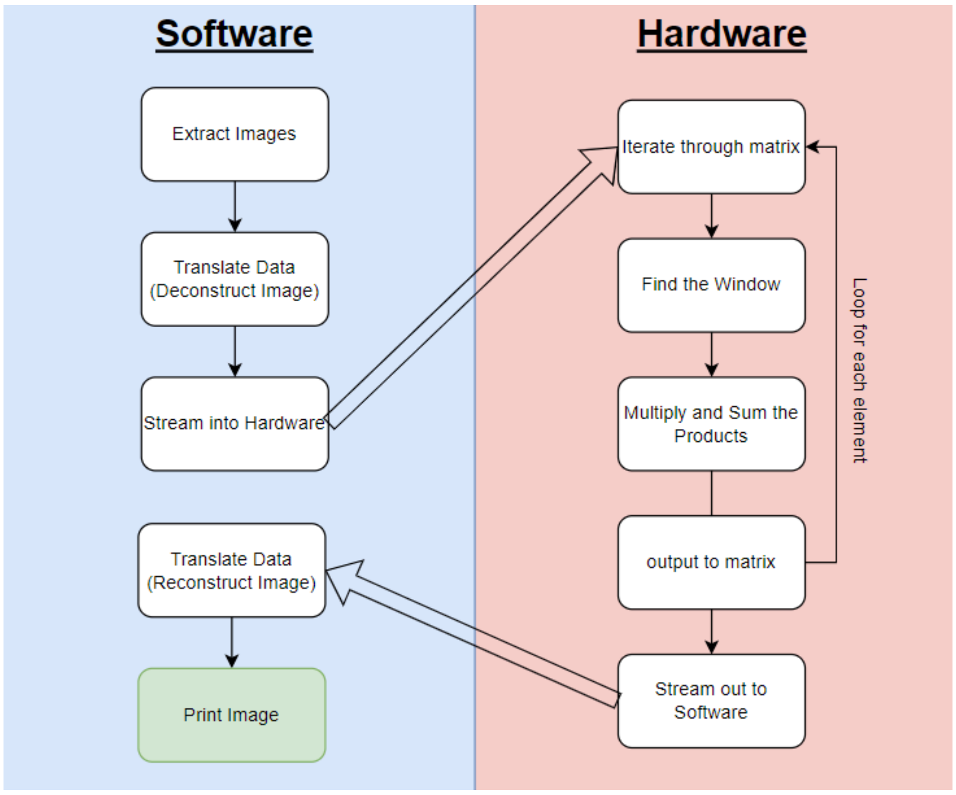
Hardware implementation

On a high level overview, the filter follows a few basic steps. The gaussian blur function in the software implementation follows these steps.

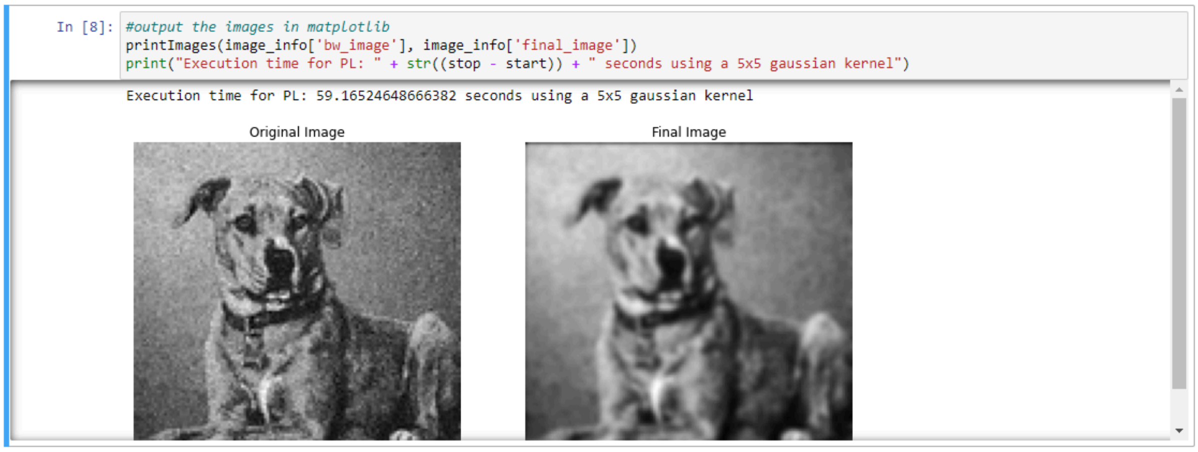
1. Iterate through the input matrix element by element.
2. Find the window (area encompassed by the kernel)
3. Multiply the window and the kernel and add the products
4. Output the sum in the output matrix.

The AXI stream interface was used as compared to AXIlite as it has a considerable advantage in bursting large amounts of data at once and no ‘handshaking’ is required. This makes AXI streaming a lot faster and more efficient than AXIlite.

There is a hardware software codesign aspect to this project. The best way to leverage hardware is by letting it operate on repetitive large values. Software is used for data manipulation and translation. In this project, the extraction of images, conversion to and from a matrix and printing the image via matplotlib will be done through software on a jupyter notebook. The hardware is responsible for executing the four steps highlighted above.



The HLS code is included with the submission. Originally, a 512x512 image ('noisy_barb.jpg) was expected to be used as the output, but as explored in the design space exploration section, the 128x128 image 'doggo.jpg' was used for comparison between the hardware and software implementations. The output of the 512x512 image can be seen in the appendix. The hardware timings and blurred image are shown below:

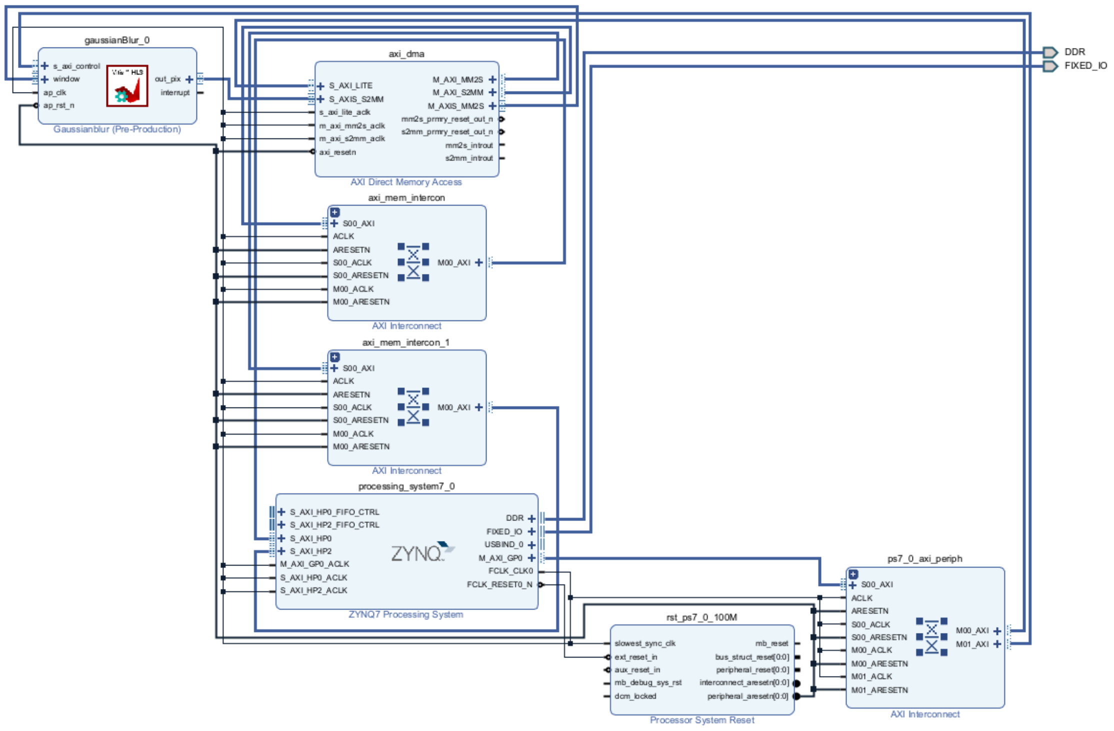


```
In [8]: #output the images in matplotlib
        printImages(image_info['bw_image'], image_info['final_image'])
        print("Execution time for PL: " + str((stop - start)) + " seconds using a 5x5 gaussian kernel")
```

```
Execution time for PL: 59.16524648666382 seconds using a 5x5 gaussian kernel
```

Jupyter notebook for the hardware implementation, the image and the bit/hwh files for the hardware demo is included in the submission. As shown by the timings above, the image ran in 59.2 seconds! Although this is slower than the 29.2 seconds achieved by the ARM chip. When taking into account clock speeds, the FPGA has a relative speedup of 3.2x! If this outcome was extrapolated, whereby the FPGA had the same clock speed as the processor of 650MHz, the total time at that clock speed would be 9.13 seconds!

Implementation Block Diagram:
AXI Stream interface was used. The DMA was adjusted to have a buffer width of 26 bits rather than the default of 14 bits to account for the large amount of data.



Design Space Exploration
A lot of design space exploration was required coming from my initial design. The base design used a 512x512 ‘noisy_barb.jpg’ image, but the synthesis predicted a BRAM usage of 1024 units. This was too high to the point where the Vivado design wouldn’t generate a bitstream. Most of the design space exploration involved reducing the usage of space on the SoC. I mostly considered the BRAM but the overall use of DSPs LUTs and FFs went down as well.

I explored using pragmas and directives first, but there was no change to my overall area usage. Loop unrolling did have an effect of reducing my BRAM usage to 0 but over exceeding the usage of DSPs, FF and LUTs. As to latency, no pipelining directive could theoretically reduce the latency of my design. This is because there is a data dependency when adding the products of the window and kernel. The addition is iterative and so inherent dependency cannot be reduced. For instance in my HLS code: output pixel sum += element in window multiplied by the element in the kernel. The sum adds the output for each multiplication. Hence, the latency cannot be reduced unless the image size was reduced completely. As shown in the table below, the 512x512 image has a latency of 262146 which is roughly equivalent to 512^2. Likewise, the 128x128 image has a latency of 16388 which is roughly 128^2. So when timing violations were highlighted in synthesis, pragmas for HLS PIPELINE off were added. So I mostly focussed on reducing the area of my designs as a way of optimising it for hardware. This is also the reason I chose to use a smaller image for the final design.

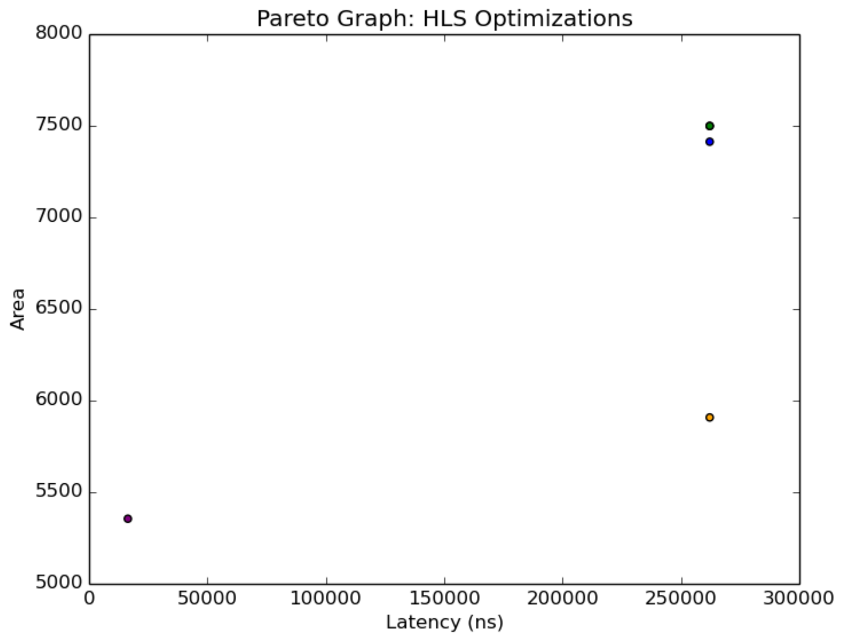
I first replaced the output matrix with a writeback matrix. This would be a smaller matrix with the same number of columns but with lesser rows. The matrix would write back the output values back into the input matrix only if the kernel does not consider the areas underneath that element of the matrix. This avoids the output pixels from interfering with the calculation of later pixels as the output pixel is influenced by the values of pixels surrounding it (above, below, to the left and right).

Next, I noticed that each grayscale pixel has a value between 0 and 255. This can be represented with 8 bits or a byte. The input and output matrices are of type int. An int is a 32 bit value (4 bytes) which wastes too much space. I used the ap_int.h library, created by AMD for Xilinx FPGAs, to define the matrices as data structures that hold 8 bit variables only. This optimization significantly reduced my area usage.

Lastly, I considered changing the image altogether. It not only reduced the number of cycles and area usage, but it also reduced the time I needed to test my IP. I integrated other two optimizations with this change in image size. I changed the image size to 128x128 pixels.

Below is my Design Space Exploration table. Graph is generated using MATPLOTLIB:

No.	Optimization	Latency	BRAM	DSP	FF	LUT	Area	Comments
1	DEFAULT 512x512 image	262146	1024	17	3575	5799	7499	
2	With Writeback	262146	512	17	3575	5799	7499	
3	With 8 bit types	262146	128	17	3557	5713	7413	
4	Writeback + 8 bit types	262146	130	5	3544	5408	5908	
5	Loop unrolling	262146	0	Too high	too high	too high	Not considered	Not considered
6	Smaller image + 8 bit types + writeback	16388	16	5	3359	4855	5355	Chosen Optimization



Conclusion

High level synthesis has many advantages, It offers a simplified way to implement a complex algorithm without using Hardware Description language. This proved to be very useful when moving my software design to Hardware as it is a lot easier to translate code from Python to C++ rather than to Verilog.

On a personal note, I was surprisingly enthusiastic about my project, and no matter how many times my design failed my tests, I immediately knew where to debug. I was also surprised by the design space exploration and curious as to why I am able to optimise certain aspects of my design, such as the area, and not others, latency. I made sure to understand why I was able to do so. Ultimately, this project allowed me to think through a hardware first perspective that will prove immensely useful when I work in the industry.

How to run:
This submission comes with a zip file that when extracted will reveal a folder called Final_Version. This folder should directly be uploaded to the SoC as it includes the bit file, hwh file, the HLS C++ file, the Jupyter notebook for the hardware implementation. To run the program:

- Make sure the folder is uploaded directly, with no changes to the output.
- Make sure that the overlay address is initialised correctly so that the notebook can pick up the correct bit file. I recommend placing the Final_Version folder directly in the jupyter_notebooks directory so that nothing has to be changed when running the notebook.

```
ol =
Overlay('/home/xilinx/jupyter_notebooks/Final_Version/dma_axis_ip_example.bit')
```

- Press run all and the notebook will go through the entire implementation and output a blurred image, along with the execution time, at the end similar to output shown in the hardware implementation section of the report.

Appendix

512x512 image blur