Hardware Accelerated Image Filtering Using PYNQ

Ross Cathcart, Kishan Maharaj, Elias Voukelatos
(202139210), (202129516), (202123675)
Project Group L
Electronic and Electrical Engineering
University of Strathclyde
EE315

***Abstract*— Field Programmable Gate Arrays (FPGA's) are finding increasing application in tasks that demand real-time processing of video and image data. The use of FPGA's in autonomous driving applications allows for image data to be processed prior to streaming the results back to the CPU [1]. This affords time saving benefits and alleviates computational pressure on the CPU, enabling it to handle other critical tasks without being burdened with excessive calculations. The PYNQ platform streamlines the development of FPGA solutions for image processing tasks by enabling productivity programming languages to be used, raising the abstraction level and lowering the barrier to entry.**

I. Purpose, Functionality & Features

Provide an overview of the purpose, functionality, and features of your PYNQ design

Typical computation tasks on image data involve filtering to extract edges, and blurring to remove high frequency artifacts. These techniques are often used as a precursor for computer vision tasks that require feature extraction such as image segmentation, object classification, and reconstruction [2].

To filter an image, a mathematical operation operation known as convolution is used. In convolution, an $n \times n$ square matrix (Kernel) with impulse response $h[n, m]$ is convolved with the image data $f[n, m]$ to produce an activation heatmap, $y[n, m]$ (Equation 1) representing the filtered image.

$$y[n,m] = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} f[k_1, k_2] \cdot h[n-k_1, m-k_2] \tag{1}$$

The choice of kernel weights determines the result of the filter, with popular kernels including Sobel X/Y and Gaussian Blur.

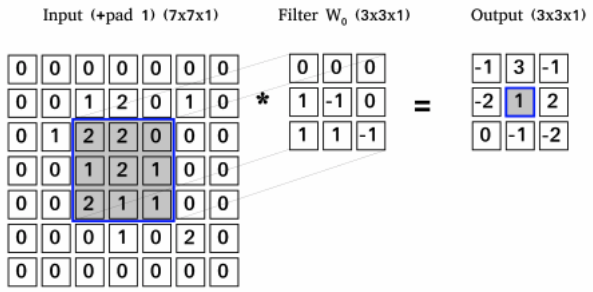

Fig. 1. 2D Convolution Example [3] - Image data (left) represented by a 2D array of pixel values. Kernel weights (centre) define filter operation. Activation array (right) is the convolution output

In this PYNQ project, the input image data is represented by a grayscale 2 dimensional array of pixels (Figure 1). Each pixel n is given by $\{n \in \mathbb{Z} \mid 0 \leq n \leq 255\}$ and represents the intensity of the pixel colour. For simplicity, image data is converted to grayscale from RGB by a simple mean average of the three channels of data. Use of grayscale in image processing is often used to achieve dimensionality reduction and facilitate faster processing where colour information is redundant [4]. For the designer, this also conveniently leads to a simpler system implementation.

The custom Overlay developed enables a custom 3×3 kernel to be passed as an argument to the IP to determine the convolution weights and thus filter action.

A key feature of the Jupyter Notebook developed is the ability for a user to numerically quantify the difference between a filter implemented fully in a software processing system (PS), to a software-hardware hybrid which makes use of the programmable logic (PL) fabric of PYNQ to accel-

erate and parallelise the calculations. Additionally, the notebook offers interactivity using widgets to allow the user to configure settings such as custom kernel and the ability to select a target image.

II. System Architecture

Review your system architecture, and explain which components are implemented in software and in hardware, and the rationale for your design choices.

Computation of a 2D Convolution is a very repetitive operation, making it well suited to parallel execution on an FPGA since each output $y[n, m]$ is a purely combinational function requiring no sequential memory of a previous state. This motivated the decision to offload this bulk arithmetic computation to the PL since this is inherently parallel. AXI4 Stream DMA was chosen for the movement of image data since it provides efficient bidirectional movement of data and frees up processor resources for other tasks [5].

Figure 2 illustrates the connections between hardware and software subsystems.

The Zynq's DDR3 Random-Access Memory is utilised both by the Zynq Processor (to manage assets flowing to-and-from the DMA) and the AXI4-DMA controller - which oversees the stream of image-processing data flowing between the custom overlay registers and main memory.

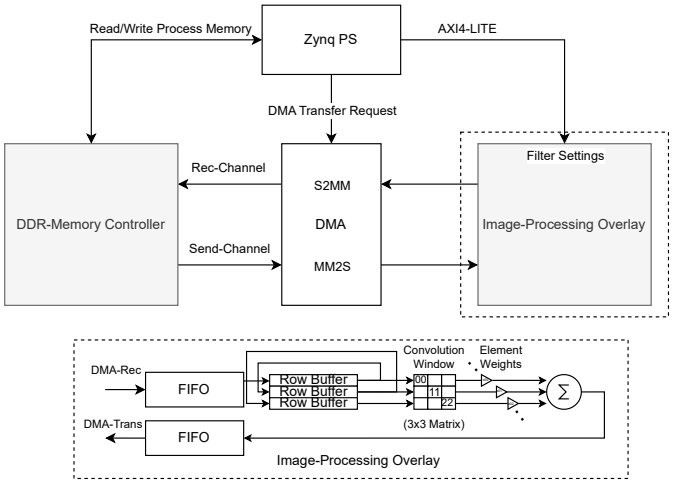

Fig. 2. Overview of Hardware & Software Architecture - These include the primary Zynq PS (ARM Core A9 microprocessor and DDR memory controller) and PL (Programmable logic and AXI4 interconnects, AXI-DMA controller, and custom image convolution IP)

The lower section of Figure 2 illustrates how convolution can be realised in a block-diagram form, and mirrors the System Generator model constructed in the project.

The image data from the DMA stream is shifted into the overlay via a one-dimensional FIFO; this data is then arranged via column and row buffers to form a window for convolution to occur.

The row buffers are used to "skip" to the next row in the pixel array to form the window by delaying the FIFO stream by the image width, which is then buffered by one element to skip to the next column. The resulting 2D array is multiplied by individual kernel weights and summed element-wise to produce the convolution output, which is fed back into the DMA stream via another FIFO.

The kernel weights, as well as other filtering settings, can be either set during hardware synthesis or at runtime using an AXI4-Lite interconnect to configure the register value. AXI4-Lite was deemed best for this application since register configuration is low throughput and does not require high bandwidth communication [6]. This makes the design flexible, but there is a trade-off between efficiency and flexibility since it also results in many unnecessary multiply-accumulate operations for kernels where the weights are zero.

The convolution calculation stage was pipelined using cut-set re-timing to reduce the significant critical path between the input and output clocked registers of the algorithm subsystem. Additionally, the CMult blocks of the algorithm were manually restricted to a wordlength of 24 bits since most common kernel weight-pixel products do not require the maximum possible wordlength format when multiplying two n bit numbers. This decision saves otherwise wasted resources on the FPGA, optimising the design for space and power efficiency. The System Generator model is shown in Figure 3.

.

III. Testing

Describe how you performed testing to ensure correction operation

Prior to the generation of the IP in Vivado, System Generator was used to simulate the design and confirm the correct behavior when compared with MATLAB's software implementation of 2D

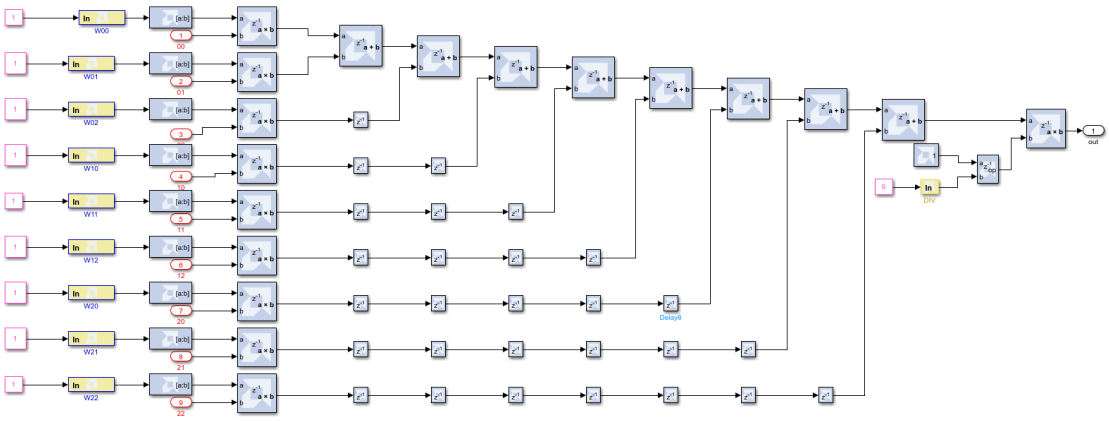

Fig. 3. System Generator Model - Pipelining has been enabled in select arithmetic blocks to facilitate faster clocking. Manual restrictions are imposed on wordlength for resource efficiency. Gateway in blocks enable weights to be configured.

convolution. System Generator features tools to support the rapid development of hardware and can allow issues in the design to be spotted early before the lengthy IP generation process. For example, an error due to unwanted truncation at a multiplier block was easily identified by inspecting the data dimensions in the Simulink Model. This example demonstrates a clear benefit of leveraging System Generator features for initial testing prior to implementation on hardware - where the diagnosis of errors is much more difficult.

As discussed in Section 2, pipelining was employed in the convolution arithmetic stage, using a single cut-set to shorten the critical path between the accumulate blocks of the convolution. Initially, this caused issues with image wraparound due to incorrect cut-set implementation - meaning not all incident paths were delayed by the same time and resulting in a "shifted" output. This effect is shown in Figure 4.

Fig. 4. Image wraparound due to incorrect cut set implementation - note the left side of the image where the right side of the image has overflown to the left.

This was fixed by adjusting the internal latency of the accumulate blocks, maintaining correct timing across all signal paths in the set. This was easily validated by observing the output image in MATLAB and zooming in at the edges to check for wraparound. After successfully simulating the filters for a range of kernels, the IP was implemented in Vivado, where the total implemented design could be bench-marked against predetermined timing and resource constraints. The utilisation and timing properties of the design are as seen in table I and Figure 5.

TABLE I

HARDWARE-RESOURCE REQUIREMENTS OF IP OVERLAY

Resource	Utilisation	Available
LUT	5168	53200
LUTRAM	1162	17400
FF	7026	106400
BRAM	10.5	140
DSP	32	220

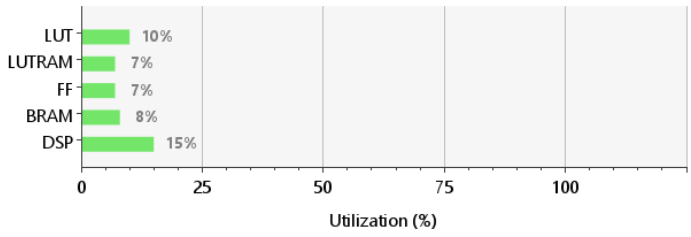

Fig. 5. Utilisation Comparison of Different FPGA Resource Blocks.

To validate the synthesised IP, the filtered image data was compared with the outputs produced using a software implementation of 2D convolution.

OpenCV provides a highly optimised version of this algorithm (`conv2d`). However, it was decided to write a custom algorithm to ensure a fairer comparison since this eliminates potential runtime optimisations (such as Single Instruction, Multiple Data [7]) which could significantly reduce the number of actual multiplications for kernels with zero weights or high symmetry. The software implementation of this 2D convolution is shown in Listing 1.

```
def software_conv(image, kernel):
    # Flip the kernel prior to convolution
    kernel = np.flipud(np.fliplr(kernel))

    # Extract dimensions of kernel and image
        data for loop bounds
    xKernShape, yKernShape = kernel.shape
    xImgShape, yImgShape = image.shape

    # Calculate the output dimensions
    xOutput = xImgShape - xKernShape + 1
    yOutput = yImgShape - yKernShape + 1

    # Initialize the output array
    output = np.zeros((xOutput, yOutput))

    # Perform convolution
    for i in range(xOutput):
        for j in range(yOutput):
            # Extract the 3x3 window
            wind = image[i:i + xKernShape, j:
                j + yKernShape]
            # Mult-accumulate
            output[i, j] = np.sum(wind *
                kernel)

    return output
```

Listing 1. Custom Implementation of 2D Convolution In Python. Numpy functions prefixed with the np namespace

Visual inspection of the software and hardware outputs facilitates a straightforward comparison of the results.

To quantify the time-saving value of the hardware accelerated filters, the IPython magic function `%timeit` [8] can be added to the Jupyter cells which execute the convolution operations on the PS and PL. Table 2 summarises the results of this comparison. The hardware-accelerated filters perform substantially better than the custom software convolution, but slightly worse than the highly optimised OpenCV implementation. This result was disappointing, since significant effort was made to optimise the System Generator model using techniques such as pipelining and wordlength optimisation. However, as was mentioned previously, this reduction in speed is a tradeoff with the kernel customisation flexibility. If more time was available, the group is confident that a number of optimised kernel-specific IP's could be produced for each filter to minimise unnecessary computations such as the four zero multiplication per $y[n, m]$ output in the sharpening kernel.

TABLE II
PS VS PL IMPLEMENTATION OF COMMON FILTERS

Filter Type	Runtime PL (ms)	Runtime PS (ms)	Custom Software Convolution
Gaussian Blur	367	251	4min 49s
Sharpen	366	163	4min 52s
Sobel X/Y	377	195	4min 50s
Box Blur	378	248	4min 54s
Ridge Detection	367	208	4min 54s

IV. Project Management And Group Work

Explain your approach to project management and group working, including the allocation of tasks, scheduling, and any challenges that arose (and how you handled them). On reflection, is there anything that you would have done differently?

An Agile approach [9] to the project management and group work was taken in the design project. Regular face-to-face group meetings and "sprints" were used to coordinate progress and work collectively on challenge areas such as the integration of the software and hardware subsystems.

Such an approach to project management was effective since it recognises that progress is often nonlinear, and flexibility to reorganise and distribute tasks continuously can be more effective than adhering rigidly to a plan.

As an example, the group encountered trouble with the implementation of flexible kernels whereby DMA communication handshakes were not be initiated properly. This was a result of passing the wrong array buffer to the AXI send and receive channels with incompatible dimensions. Rather than one group member tackling this problem independently, the group met collectively to work on the problem and re-evaluate task distribution to reflect the temporary progress stall.

The initial phase of the project required acquiring a suitable mathematical foundation for image processing, including 2D convolution. Subsequent development of the software, hardware, and documentation was conducted in parallel before finally integrating all features within Jupyter. Figure 6 illustrates the project workflow.

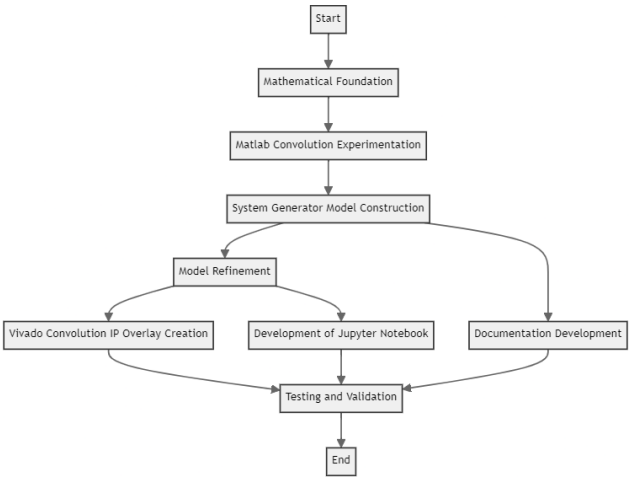

Fig. 6. Project Workflow - Parallelism in tasks reflects the Agile approach to project management, since it enabled team members to work on different aspects of a project concurrently, increasing efficiency and reducing overall project duration

On reflection, this approach was successful and worked well for a small group setting but more formal planning methods such as the production of a Gantt chart would likely be more useful to organise a larger project. In such projects, it may be difficult to ensure that all tasks are being completed on time since regular full team meetings may not be possible.

V. Project Takeaways

Reflect on what you have learned from this project, highlighting aspects that have required independent research and investigation

This project has provided an introductory insight into applied image processing and computer vision, highlighting the potential of FPGA's to revolutionise the field by accelerating computationally demanding tasks. In addition to applying theoretical knowledge from EE315, such as pipelining to optimise a digital design, further reading was necessary to acquire the necessary technical background.

In particular, a filtering technique known as 2D convolution was independently explored since this forms the backbone for much of image processing. The concept of a Kernel as an impulse response was new to the group and an interesting extension to the familiar one-dimensional convolution in the context of signals and systems.

Further to acquiring a mathematical appreciation for this operation, research was also required to understand how to translate the algorithm into hardware and software efficiently. This forced the group to consider practical design decisions, such as the tradeoff between increased resource utilisation (wordlength) and speed of the design.

The Jupyter environment was largely unfamiliar to group members, so consultation of online documentation for advanced features such as widgets, HTML user elements, and LaTeX integration was necessary to produce an interactive and attractive notebook. This project highlighted the usefulness of Jupyter in enabling repeatable and transparent research to be conducted and then easily shared with others.

References

[1] D. G. Bailey, "Image Processing Using FPGAs.," *Journal of imaging*, vol. 5, no. 5, 2019.

[2] S. Kim, C. Song, J. Jang, and J. Paik, "Edge-aware image filtering using a structure-guided CNN," *IET Image Processing*, vol. 14, no. 3, pp. 472–479, 2020.

[3] European Space Agency Machine Learning Group, "*Convolutional Neural Networks: Introduction*." Available at `https://www.cosmos.esa.int/web/machine-learning-group/convolutional-neural-networks-introduction` (Accessed: 2024-04-06).

[4] K. Okarma and J. Fastowicz, "Computer vision methods for non-destructive quality assessment in additive manufacturing," in *Progress in Computer Recognition Systems* (R. Burduk, M. Kurzynski, and M. Wozniak, eds.), (Cham), pp. 11–20, Springer International Publishing, 2020.

[5] AMD Corporation, "*Processing System 7 v5.5 Product Guide (PG082) .*" Available at `https://docs.amd.com/v/u/en-US/pg082-processing-system7`. (Accessed: 05-04-2023).

[6] Florentw, "AXI Basics 1 - An Introduction to AXI." Xilinx, February 2023. (Accessed : 7/04/2024).

[7] OpenCV Contributors, "Vectorizing your code using universal intrinsics." Available at : `https://docs.opencv.org/4.x/d6/dd1/tutorial_univ_intrin.html` (Accessed 24/04/2024).

[8] "timeit – measure execution time of small code snippets." `https://docs.python.org/3/library/timeit.html`. (Accessed: 5/04/2024).

[9] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," *CoRR*, vol. abs/1709.08439, pp. 13–20, 2017.