ET4370 - Reconfigurable Computing Design Lab. Assignment Adding Grayscale Contrast Enhancement functionality to an existing Simple Video Processing Pipeline implementation on FPGA

Group 5

Pradeep Venkatachalam (4735323) & Anurag Kulkarni (4627342)

February 5, 2018

1 Introduction

The goal of this assignment is to implement a video processing functionality on an existing simple video processing pipeline on FPGA. Xilinx's Vivado Design Suite is used to perform a High-Level Synthesis of the chosen algorithm. The hardware development platform used for the lab. is Xilinx's PYNQ-Z1, which hosts a ZYNQ XC7Z020-1CLG400C SoC containing an ARM Cortex A9 processor and a Artix-7 family programmable logic[5].

1.1 Background

PYNQ is an open-source project aimed to achieve higher productivity in the design of embedded systems. PYNQ stands for Python Productivity for ZYNQ. The project offers a technology called *Jupyter Notebook*, which is a browser based interactive computing environment and uses Python to invoke hardware libraries (overlays) to implement a functionality in hardware (i.e., on the FPGA)[1]. PYNQ also comes with open-source packages like OpenCV and Numpy, which ease the task of programmers developing image processing algorithms. Thus, Python, here, acts as a glue code to interface to hardware and as a language to develop programs to run on Linux running on the ARM processor. The *ipython* kernel is a computation engine that executes the Python code, and is thus said to be called by the Notebook(s).

Hardware libraries are meant to speed up the execution of suitable applications (like streaming applications) as they are hardware implementations, capable of massive parallel processing and waiving off the software-hardware interaction latency due to temporal processing. The given pipeline inputs an HDMI stream pixel-by-pixel and outputs the same stream, thus qualifying as a streaming application. Our intended algorithm is expected be a (hardware) functionality block inserted within the pipeline to perform the required image processing. The same algorithm is implemented in software (Python) meant to be run on Linux running on ARM. In section 2 we state our understanding of the provided video processing pipeline's hardware implementation. Details of the chosen algorithm and its justification are provided in Section 3. We compare the results obtained for our software and hardware implementations and also state our findings about the limitations in Section 4.

2 Our understanding of the provided basic Video Processing Pipeline

The Video Processing Pipeline code provided makes use of the `Video` subpackage from PYNQ. The Video hardware subsystem consists of a HDMI-In block, a HDMI-Out block, and a Video DMA[2], as shown in Figure 1.

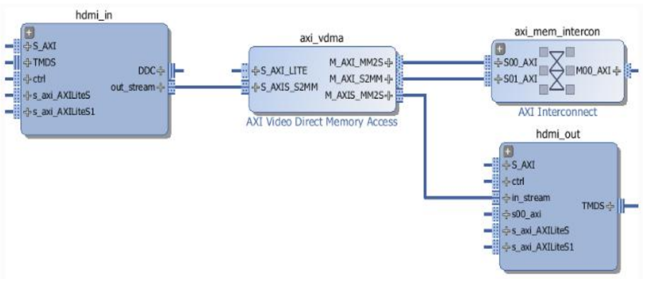


Figure 1: Block diagram of the Video hardware sub-system[2]

A modified pipeline processing is carried out on the Video pipeline. The HDMI signal blocks are AXI-stream[2]. A custom AXI-stream IP with an input stream and output stream is inserted into the video pipeline after the pixel_pack block on the HDMI-In block. The port type `s_axilite`, which is a light version of the AXI bus (an on-chip interconnect specification), is used on input side of the blocks which accept the AXI stream, which is used as the video stream. The `stream` IP block provided routes the stream to DRAM through the `AXI Video Direct Memory Access` block. The corresponding block diagram is shown in Figure 2.

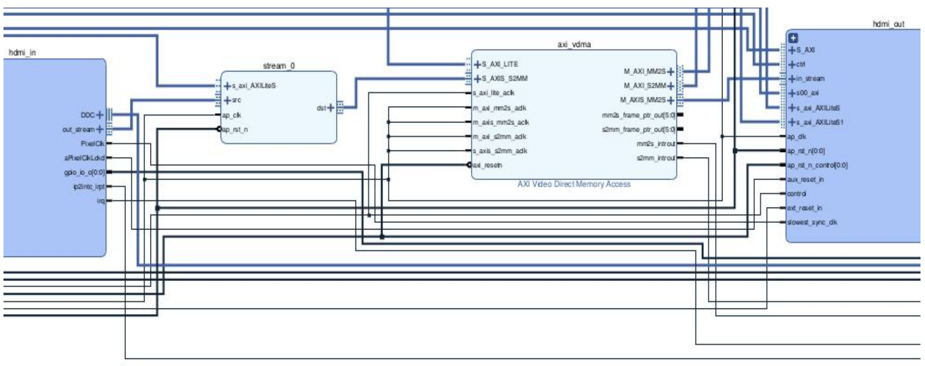


Figure 2: Portion of the block diagram of the basic video pipeline implementation provided between hdmi_in and hdmi_out

The `stream` block is the hardware implementation of the *stream* function in the `main.cpp` file of the provided code after HLS is performed. From a software point of view, this function is called each time a new pixel is read (of course, after the previous one has been outputted). In the provided code, only a suitable conversion to 24-bit color space is performed for the screen connected to the output HDMI. Thus, it is inside this function that some processing was deemed to be performed.

3 Grayscale conversion and Contrast Enhancement - justification of choice

As explained in Section 2, a suitable algorithm for stream processing was required, and which also would not require storing an entire *linearized* frame into the DRAM, which is inadequate for such operations. Thus, the task of grayscale conversion of the input frame and performing contrast enhancement on it was taken under consideration. The algorithm is explained below.

Step 1: Convert each pixel to grayscale using the formula:

```
gray[i,j] = 0.3*color[i,j,2] + 0.59*color[i,j,1] + 0.11*color[i,j,0]
```

The order of 8-bit color value locations (indices) for each 24-bit pixel data was 0 for red, 1 for green, and 2 for blue. Thus, the gray pixel color value required 8 bit storage.
Step 2: Update histogram based on the gray level of each pixel. This histogram holds for an entire frame.
Step 3: Modify the gray pixel value in the following manner:

```
if (gray[i,j] < threshold_min):
        gray[i,j] = 0
else if (gray[i,j] > threshold_max):
        gray[i,j] = 255
else:
        gray[i,j] = threshold_max*(gray[i,j] - threshold_min)
                    /(threshold_max - threshold_min)
```

Step 4: Calculate threshold_max and threshold_min for each frame to be used by the next frame.

```
x = 0
while(x < HISTOGRAM_SIZE and histogram[x] < CONTRAST_THRESHOLD):
        x = x + 1
threshold_min = x
x = HISTOGRAM_SIZE - 1
while(x > mini and histogram[x] < CONTRAST_THRESHOLD):
        x = x - 1
threshold_max = x
```

where HISTOGRAM_SIZE = 255 (largest gray value) and CONTRAST_THRESHOLD = 80 (found through arbitrary experiments).

The entire operation per frame mentioned in the steps above can be summarized as per Figure 2 from a histogram perspective.

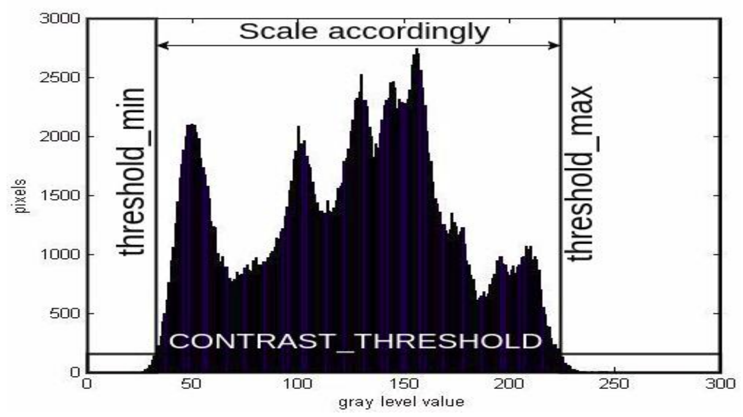


Figure 3: Contrast enhancement operation for a grayscaled frame from a histogram perspective

The threshold_max and threshold_min values are 255 and 0 respectively by default. This means that for the first frame, no thresholding and contrast enhancement are actually performed as 0 and 255 are the limits of the histogram for 8-bit gray values, but the subsequent frames will be contrast enhanced through thresholding with thresholds calculated based on previous frame. The result would not be visually very different due to the high coherence likelihood between consequent video frames captured at 60 fps. This eliminates the requirement for storing an entire frame into the DRAM, only the thresholds (and the 256 byte histogram) needed to be stored per frame operation for the next frame in so-called registers during the hardware implementation. Thus, this algorithm was selected for HLS.

4 Performance Analysis, Results and Limitations

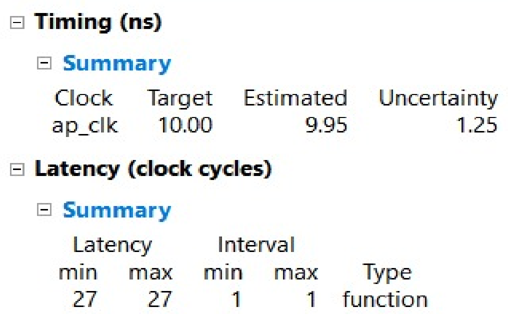
Average Latency per Frame was chosen as a metric for performance analysis of both our software (Python processing on the Processing Subsystem of PYNQ) and hardware (HLS) implementations of the algorithm, since video processing performance often indicates how fast the frame was processed. Besides, the hardware implementation was meant to speed up the task due to its parallel processing capability, and to compare its performance with our software implementation, this metric was suitable. The input source was the webcam of a laptop, which generated frames of sizes 480x640 @ a fixed frame rate of 15 fps, hence *throughput* would not be correct indicator of performance. The output was rendered on a TUD monitor. It is noted here that the `base` video overlay configures the input and output HDMI blocks at 1280x720 (the software implementation uses the same bitstream, and the bitstream for the hardware implementation does not modify this resolution). Thus, the output is available at the same resolution, with portions not part of the input frame blackened.

4.1 Software Implementation

The software implementation resulted in an average latency per frame of **418.23-7366 seconds**. Since this is not an acceptable figure for real-time processing, parts of the code consuming more time were identified. The components of the implementation consuming the highest amount of time were histogram computation and threshold calculation through traversing the histogram, which together consumed **418.0233886 seconds**, i.e, 99.95% of the latency per frame on an average. Histogram computation is inherently a sequential operation, and so is its traversal for calculating the thresholds. An approach to these challenges is discussed in the next subsection.

4.2 High Level Synthesis

FPGAs can result in optimized hardware implementations for histogram computations [3]. The high level compiler was believed to generate optimized modules for histogram and threshold computation, which could operate on portions of the histogram in parallel at different clock frequencies and utilizing duplicated memory ports to avoid memory collisions[3]. The pragma `HLS PIPELINE II=1` was used to allow concurrent operations through pipelining, processing a new input every clock cycle. The implementation however, did not work, resulting in multiple images to appear on the output screen. A possible reason was cited as the processing time per pixel still being too long with the overlay confused about the information of the first pixel of the next frame (i.e., the `user` signal being asserted incorrectly). This was also warned in the synthesis report where the latency reported a question mark, indicating the bounds of latency of the deigns were unknown[4]. The histogram and threshold computation was then removed from design, and fixed thresholds, based on arbitrary experiments were used as `static` variables, with `threshold_max` being 250 and `threshold_min` being 15. The synthesis report for this design is shown in Figure 4.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.95	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
27	27	1	1	function

Figure 4: Synthesis report for the hardware design with fixed thresholds

As per the report, the estimated latency is bounded by 27 clock cycles, thus the average latency per (1280x720) frame could be estimated as **0.2488332 seconds** (clock period = 10 ns), which is comparable to the grayscale conversion only software implementation (no thresholding) latency of **0.2139774 seconds**.

5 Conclusion

Due to the limitations discussed in section 4.2, the final hardware implementation for grayscale conversion with contrast enhancement is presented only with fixed thresholds, i.e., without the histogram computation for the gray converted pixels. Our finding is that explicit code for efficient histogram processing should have been written (instead of the simple array handling), and techniques such as those mentioned in [3] should have been implemented in high level.

6 References

[1] Jupyter notebook documentation. `http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html#kernel`
[2] PYNQ documentation. `http://pynq.readthedocs.io/en/v2.0/pynq_libraries/video.html#pipeline-processing`
[3]`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.8766&rep=rep1&type=pdf`
[4] Xilinx High Level Synthesis documentation. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/ug902-vivado-high-level-synthesis.pdf`
[5]The PYNQ official webpage. `http://www.pynq.io/`