1. Introduction
This project implements an algorithm to perform real-time Corner Detection on a high definition video stream. Our algorithm is executed on a Xilinx ZYNQ evaluation board using C++ and Vivado HLS. Corner detection is the fundamental task in many computer vision and image processing applications. The corner detection algorithm filters an input color image and outputs four corners of an image with black background and corners in white. A good Corner detector should have following properties 1) detects true corners - it should detect all true corners and no false corners, 2) good localization - corners are found as close as possible to the real corner in the image, 3) robust to noise - corners are only detected once and the algorithm should not respond to image noise. Median filter is used to remove the background noise to improve the results.

In this project, we demonstrate a marked improvement in execution time of the real-time corner detection through a hardware implementation of the algorithm over an OpenCV-based software implementation. We implemented corner detection algorithm through a dataflow which utilizes OpenCV-like HLS constructs and C++ functions. Using the Vivado HLS flow, we were able to convert our C++ code into synthesizable RTL which we mapped onto the Artix 7 based FPGA on the ZYNQ board. Our design is a fully pipelined design with an Initiation Interval of 1. Using a standard HDMI cable, we can input real-time video into the ZYNQ and show real-time corner Detection of our output on a screen using either the software implementation or the hardware implementation. Our hardware corner detector achieves a clock frequency of 148.5 MHz to yield each frame of 1920x1080 pixels. There is no down sampling in our design.

2. Algorithm Overview
The Corner Detection algorithm is structured in a way that lends itself to pixel-by-pixel streaming data flow of the video. The algorithm consists of four main stages. Almost all computations on a pixel can be completed on that pixel individually, or within a small neighborhood of pixels around the pixel in question. Line buffering and memory windows are used in our implementation to store the pixels around the current pixel being processed such that it is not necessary to ever obtain a pixel more than once in a certain stage of the algorithm. This way we can stream the video in 1 pixel at a time, and never have to store the entire frame of the video. This dataflow organization allows us to achieve a throughput of one pixel per clock cycle, reduces the need for large memory structures, and facilitates real-time video processing. Moreover, there is no need to down sample the input stream and all the computations are done at speed.

This section contains different stages of implementation:

1. **RGB stream to YCbCr conversion and red color thresholding** - This block takes AXI stream from input source and converts RGB input to YCbCr format. After that it does thresholding on the YCbCr stream pixel by pixel to separate red component in the stream. This output stream now goes to median filter for noise reduction in the image. It will give a stream of 1’s and 0’s based on the whether pixel is red or not.
2. **Noise(Median) Filter**- The median filter is used to reduce the noise in an image. The median filter considers each pixel in the image in turn and looks at its neighbors to decide whether or not it is representative of its surroundings. It simply replaces the pixel value with the median of neighboring pixel values. The median is calculated by counting number of ones in the window and then replacing the pixel being considered with the middle pixel value. A 5x5 window median filter is used in this design and it prevents the detection of salt and pepper noise as a corner and gives de-noised image as output without blurring effect.

3. **Corner detect**- Corner detect block takes output stream of median filter block as input and detects four corners in a video frame. These calculated values are passed to the display corners block that displays the output stream along with these coordinates of corners.

4. **Display corners**- This block takes four corners coordinates computed from corner detect block and assigns white color values to these coordinates. Rest of the pixels are assigned black color in the output stream. The output of this block goes to AXI stream conversion block that converts into output at the screen.

3. **Implementation**

Implementation of our algorithm is based on the app note provided by Xilinx (Ref [1]). The app note provides a sample implementation to enable communication between HDMI video input and HDMI video output on FPGA. The FPGA board used for this project is Xilinx ZYNQ FPGA which provides AXI bus interface to take HDMI input, process it using either FPGA or on-board ARM processor and stream it as HDMI output. Our implementation is able to execute hardware version of the corner detection algorithm. The hardware version of algorithm is the one that gets synthesized into RTL code and uploaded to FPGA as bitstream. The software version of the code is a reference implementation that gives the expectation and visualization of corner detection algorithm on a static image. The programming language used for both the versions is C++. The hardware version utilizes third party HLS libraries from Xilinx that replicates some of the behaviors of OpenCV libraries. The software version is purely implemented in OpenCV for the reference.

The architecture of the corner detection block is shown in Figure 3.1. The parent block is divided into 6 individual sub blocks that are described in previous section. The entire architecture follows dataflow model wherein all the blocks start executing as soon as pixel is available at input. The AXIvideo2Mat block waits for video input from HDMI port and the moment it gets the frame pixel, it converts the video into Mat representation and it begins the execution of dataflow model. Note that the reason for using Mat representation is to give user an illusion of implementing OpenCV functionality, but underneath it uses totally slightly data structure than OpenCV version of Mat. After the dataflow execution starts, RGB2YCbCr block takes that RGB format video stream and starts converting it into YCbCr format pixel-by-pixel. This module is mandatory because YCbCr format is less prone to surrounding noise. So, be it dark or light red, it will be detected as red and we will easily be able to segregate the object of interest. To reduce the resource utilization, we converted 3 channel YCbCr image to single channel grayscale image. Once the thresholded pixel is calculated by YCbCr block, it goes to median filter block. The median filter block observes all the previous 5x5 pixels and gives the output based on the pattern of these pixels.
To minimize the resource utilization, median filter is using line buffer concept as shown in Figure 3.2. Instead of storing entire image stream into buffer and then applying median filter, the line buffer concept only stores previous 4 rows of the frame. The window gets these 4 rows from line buffer and 1 row from input pixel and calculates the number of 1s inside the window. If the count is greater than threshold, it gives 1 as output, otherwise it gives 0 as output. After streaming out the video pixel, window buffer shifts right and repeats the process until the frame is finished.

The result of median filter is then given to the corner detect block. Corner detect block determines the corner of isolated red object by simple max/min logic. It finds the extreme 4 points of the video frame by determining the points corresponding to Xmin, Xmax, Ymin and Ymax as shown in Figure 3.3. This algorithm gives proper results when the frame is already known in prior. But in our case, as we were following dataflow pixel-in pixel-out model, it was difficult to predict the final corners of the frame unless we reach to the last pixel of the frame. So to overcome this limitation, we decided to stream-out the corners of previous frame while current frame is busy detecting the corners. So in reality, the corners, currently shown on the screen, will correspond to the previous frame of the original video and the current frame’s corners will be shown on the next output frame. As HDMI video renders at 60 frames per seconds, the visual difference of 1 frame lag is negligible. The job of next display corner block is to get corners from previous block and using it to give output stream. The region where corners exists will be displayed as white dot and remaining area of the video frame will be black colored. For testing the functionality of the architecture, we have considered a “test_1080p.bmp” image as input. Then we are executing this logic function (which is image_filter in code) to detect the corners. The result image is stored in “result_1080p.bmp”. It’s important to note that first call for image_filter will give all corners as 0 (because it gives previous corners, which are zero). It’s the second call for image_filter which gives properly detected corner. That’s the reason why we have called image_filter function twice in the test-bench that we have considered.
The software implementation of the corner detection is implemented by utilizing OpenCV libraries. OpenCV provides support for some libraries to do Color conversion and image thresholding. By using these libraries and writing custom defined OpenCV corner detection algorithm, we are able to detect four points of a static image. But due to a technical challenge, somehow it’s not being able to plot all the corners as output image. As a result only half of the corner detected image is stored as output file. The output of software version of corner detection logic is stored as “result_1080p_golden.bmp”.

4. Evaluation:
Shown below in Figure 4.1, 4.2 and 4.3 is an example image as it is pushed through our algorithm’s data flow. This shows the image after every intermediate step until we display all the four corners of a red colored object in the image that is shown in Figure 4.4.
The figures shown above indicate the way in which this project was tested in simulation. To test whether the result is right or not, we had an equivalent CPP code wherein we made use of similar opencv functions. We indicated whether the hls code is right or not by comparing the results from the opencv and hls code. Once we verified that this design works under simulation, we generated the bitstream and tested the same on the FPGA.
The resources occupied by this functionality on the FPGA is shown below:

<table>
<thead>
<tr>
<th>Functionality</th>
<th>II</th>
<th>BRAM (%)</th>
<th>FF (%)</th>
<th>LUT (%)</th>
<th>DSP48E (%)</th>
<th>Est. Clock (Target = 6.67 ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB2YCb Cr</td>
<td>1</td>
<td>0</td>
<td>~0</td>
<td>~0</td>
<td>~0</td>
<td>5.09</td>
</tr>
<tr>
<td>Median</td>
<td>1</td>
<td>1</td>
<td>~0</td>
<td>1</td>
<td>0</td>
<td>5.71</td>
</tr>
<tr>
<td>Corner Detect</td>
<td>1</td>
<td>0</td>
<td>~0</td>
<td>~0</td>
<td>0</td>
<td>6.27</td>
</tr>
<tr>
<td>Display Corner</td>
<td>1</td>
<td>0</td>
<td>~0</td>
<td>~0</td>
<td>0</td>
<td>5.90</td>
</tr>
<tr>
<td>image filter</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>~0</td>
<td>6.27</td>
</tr>
</tbody>
</table>

Our design is making use of 1% BRAM resources, 5% LUTs and almost 0% DSP48Es. The entire design is pipelined with an II (Initiation Interval) of 1 (i.e) the design can take in a new pixel input at every clock cycle. The maximum achievable clock period is 6.27 ns where the median block sets this cycle time. The design was tested with a clock period of 6.67 ns (150 MHz) clock period.

Executing this code on the ARM core avoids the usage of programmable logic blocks. However, implementing it on the ARM core makes it 10x slower and also it is very difficult to make it work for real-time video. This re-enforces the need for a dedicated hardware block for such computation intensive applications. In this design, we have adopted a simple four point corner detection algorithm. There are various other accurate corner detection algorithms like Harris corner detection which would provide more accurate results. Also there are many other algorithms which can detect multiple corners (not limited to four). However these algorithms involve huge computational tasks which may occupy higher number of resources compared to the current design. Also the computations involved in these algorithms may increase the cycle time and thus reduce the clock period. Also for a given clock period, these advanced algorithms might not be able to achieve an II of 1. Thus our current design is a trade off between the design complexity and achievable throughput.

5. Project Management

The timeline chart below shows how the overall project was subdivided into tasks and how these tasks were split among the three individuals in the group.
Once we were asked to come up with a project topic, we took a week’s time and finally we thought about implementing real-time corner detection algorithm on Zynq board. We then took a week’s time to do some background research on how we are going to proceed with this project. We came up with a rough block diagram of what we wanted to implement. Instead of implementing the entire algorithm on the whole, we took an incremental approach wherein we first focussed on thresholding an image with respect to the target color (red) and observed if simulation is working correctly for this block. We ensured that our C++ code is synthesizable and can fit into the FPGA by running the Vivado HLS synthesis and by verifying the clock requirement and resource usage, as and when we added a new module/function in the overall design. We followed the same approach for adding median filter and corner detection modules and functions. During this period, we faced many challenges wherein we faced some setup time violation errors as the clock period of 6.67 ns could not be supported by few modules. We had to either reduce the clock frequency or increase the initiation interval of the pipelined design. Since we didn’t want to compromise on both, we tried and succeeded in modifying the modules’ logic.

Once we got the entire design working under simulation, we generated the bitstream and tried to test it on the FPGA. We fed the HDMI input through a webcam and were viewing the output on a HDMI compatible monitor. On our first try, we could make the code run only on the ARM core. We couldn’t make the synthesized programmable logic blocks work because we were passing the input image to the “Display Corners” block shown in Figure 3.1 though that block doesn’t make use of it. The communication between each and every block takes place through FIFOs. Hence in this case, the frame coming out of “Corner Detection” block was flooding the input FIFO of “Display Corners” block before the “Display Corners” block could receive the four required corner inputs. This in-turn lead to deadlock. After detecting this issue with Professor’s help, we modified the logic so as to avoid passing the input image/frame to the “Display Corners” block. This helped us in getting the four corners displayed in real-time using hardware/FPGA. However, the output was still flickering and the first input video frame was still appearing in the background.

### 6. Conclusion

This project gave us experience in high level synthesizable C++ code that can be efficiently used to implement the design with the Vivado HLS tool. We learned how to create a streaming video processing architecture, such as performing data-flow pipelining and using line buffers and memory windows to
store data for video processing. We also gained insight into image processing and corner detection algorithms and different types of filters used for processing noise in the images and the C++ based OpenCV library. Finally, we learned how efficient it is to design hardware accelerators and coprocessors using C++ and HLS tools rather than low level RTL. The cycle time is very short when using high level language as most of the optimizations are done by the tool like area and timings. Moreover the design runs much faster on FPGA as compared to ARM core. However, debugging the design can be inefficient sometimes when the design does not follow correct data-flow or control flow. Overall, this project allowed us to experience first hand how to implement an algorithm in hardware through the use of high level synthesis.

7. References


(3) Xilinx HLS Video Libraries: http://www.wiki.xilinx.com/HLS+Video+Library

(4) Xilinx design suite user guide:

(5) Implementing Memory Structures for Video Processing in the Vivado HLS Tool:

(6) Median Filter and Sorting Network for Video Processing with Vivado HLS:
    http://ens.ewi.tudelft.nl/Education/courses/et4351/Median.pdf

(7) ECE 5760 Project: Bruce in a Box:

(8) ECE 5775 Project: Fall-2013 - Canny Edge Detection:
    https://dib10bmlvqabco.cloudfront.net/attach/hyat4pjpbnz1a6/h91gmb7lry4ux/i3fyqo4ljxm/canny_ece5775_f13.pdf