A CIRCLE HOUGH TRANSFORM IMPLEMENTATION USING HIGH-LEVEL SYNTHESIS

By

Carlos Lemus

Bachelor of Computer Engineering
University of Nevada Las Vegas
2018

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Engineering - Electrical Engineering

Department of Electrical and Computer Engineering

Howard R. Hughes College of Engineering

The Graduate College

University of Nevada, Las Vegas
December 2020

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 24, 2020

This thesis prepared by

Carlos Lemus

entitled

A Circle Hough Transform Implementation Using High-Level Synthesis

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Engineering - Electrical Engineering
Department of Electrical and Computer Engineering

Emma Regentova, Ph.D.
Examination Committee Chair

Venkatesan Muthukumar, Ph.D.
Examination Committee Member

Mei Yang, Ph.D.
Examination Committee Member

Yoohwan Kim, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Abstract

Circle Hough Transform (CHT) has found applications in biometrics, robotics, and image analysis. In this work, the focus is the development of a Field Programmable Gate Array (FPGA) based accelerator that performs a series of procedures and results in circle detection. The design is performed using Vivado High-Level Synthesis (HLS) tools and targeted for a Zynq UltraScale+ ZCU106. The implementation includes the following procedures: Gaussian filter, Sobel edge operator, thresholding, and finally the CHT algorithm. The performance is evaluated based on the execution time as compared to the software (Python code) execution and the analysis tools provided by Vivado HLS tool. The accuracy of detection is evaluated due to the approximation done for the sake of faster execution. The CHT requires a large amount of memory for its implementation, and thus the overall resource utilization is to be optimized. In this work we evaluate both the speed (time) and the number of logical blocks and memory components required for implementation. The core of the work is the efficient implementation of the Circle Hough Transform using High-Level Synthesis.

Keywords: Circle Hough Transform · High-Level Synthesis · Circle Detection · Field Programmable Array

Acknowledgments

First, I would like to thank Dr. Regentova for being an incredible mentor and a pillar of support throughout my entire collegiate career. It has been a long journey to get to where I am at right now and it is because of her help and guidance that I have been able to accomplish my goals.

I would also like to thank my friends and family for being a constant source of encouragement. Particularly, my friends Brandon and Prentyce for letting me pick their brains to workout solutions during this process. Finally, I would like to give an incredibly special thanks to my parents, Axel and Lissette, and my sister, Daniela, for keeping me on track and being the reason that I have made it this far.

Thank you

Carlos Lemus

University of Nevada Las Vegas,
December 2020

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Automatic circle detection is an important part of extracting information in computer vision. Circle Hough Transform (CHT) is one of the most popular algorithms for circle detection due to its tolerance to noise. It has been used in various fields including biometrics, robotics, and mobile applications.

This work proposes a circle detection implementation of the CHT algorithm using Vivado High Level Synthesis (HLS), an Integrated Development Environment (IDE). The tool provided by Xilinx allows for automated compilation of high-level language code (i.e., C, C++) to hardware development languages, such as VHDL or Verilog. Cores can be synthesized and integrated using Vivado Design Suite to be utilized in embedded applications. In this study, we target the Zynq UltraScale+ ZCU106 development board because of its high-performance speed and resources, such as Look Up Tables (LUT) and Processor-less Block Ram (BRAM).

A Gaussian filter, Sobel Detection, and thresholding procedures are utilized as preliminary steps to automatically detect circles in imagery. All three of the functions mentioned have been implemented by Xilinx in an xfOpenCV library similar to that of OpenCV. However, these functions have been accelerated using methods of parallelism only available to hardware, such as unrolling of loops, pipelining, and partitioning and reshaping arrays to improve speed and memory.

The speed that Programmable Logic (PL) brings to this step can then be coupled with procedures on the Processing System (PS) to perform more complex operations, such as execution of machine learning techniques and use of the CHT as a part of its post-processing procedures. In other words, as a component of the whole. The CHT algorithm implemented in this work is not one that can be found in the xfOpenCV library. In this case, we explore the accuracy and performance of the high-level synthesis algorithm as the groundwork of a larger project. This can later be implemented on an FPGA using real-time video and software co-design to accelerate circle detection methods for embedded applications.

The rest of the work is organized as follows: Chapter 2 discusses the related work. Chapter 3 describes the proposed CHT implementation and the circle detection process. Chapter 4 describes the experiments and results. Finally, the work is concluded in Chapter 5.

Chapter 2: Related Works

There have been several applications of the CHT algorithm that take advantage of its parallelism and result in real-time solutions. Some mobile devices which implement the CHT algorithm have been developed to accurately segment the iris in the biometrics field [6,10]. These designs are in demand so that they can be easily integrated into mobile personal or commercial applications and connected with other hardware. In these applications the hardware implementation is the only solution, such as one that is embedded into the iris detection and scan for security. The problems, such as detecting wheels and road signs [4], are among just a few for mobile applications. This excels over a software implementation as embedded solutions almost always provide a real-time performance upgrade.

There have been several different methods proposed for effectively detecting circles. Methods based on the CHT are widely used due to their effectiveness, however, it requires huge computational and memory requirements. Some have sought to improve this with new implementations such as using an incremental CHT (ICHT) and a CORDIC algorithm that utilizes the parallel properties to decrease computation time and manage resources. Djekoune *et al.* presented a method of reusing the previously computed circle point coordinate values to derive the next circle point removing the trigonometric functions needed during the CHT voting process [3]. This greatly reduces the computing time required to achieve CHT.

Other works have shown that a gradient of the circles can be used instead of the original method of using trigonometric functions for the implementation of CHT. Chen *et al.* [7] With the rising popularity of machine learning methods, studies have also shown ways to detect circles, such as using a population-based system based on the Teaching Learning Based Optimization algorithm [2].

Many architectures and algorithms have been proposed to accelerate the CHT algorithm [4]. One of these methods is on a Field Programmable Gate Array (FPGA), an embedded platform that has risen in popularity as a method to produce accelerated hardware products. In most recent works, Kumar *et al.* proposed a memory efficient CHT implementation on an FPGA for the implementation for iris localization. It utilizes 320x240 sized images and an assortment of 1D and 2D accumulator arrays to find circle centers and reduce the memory required resulting in fast processing times. It also investigated that to make the circle detection more accurate the image must be preprocessed to reduce false edges before applying CHT [11].

While the section has discussed several existing CHT architectures, with a focus on FPGA, they can be modified and implemented on HLS. This allows for a for quick production of Intellectual Property (IP) cores for embedded systems that can speed up advanced algorithms and accurately measure the speed and memory usage of the implementation. Further work would require minimal work to the structure of code and algorithm instead of the hardware design achieving similar results to its counterparts.

Chapter 3: Circle Detection

Image processing contains several steps to prepare the image for detection. Images must be enhanced, or features must be first extracted from the image itself. In this section, we describe the steps required to extract features necessary for the CHT algorithm to detect circles. Many of the preprocessor steps are implemented using OpenCV for Python and xfOpenCV for Vivado HLS as they are already optimized. However, the hardware implementation is discussed in the text. As a first step, the input images of size MxN are converted to the gray intensities.

3.1 Circle Detection Procedure

First, color images assumed to be of the size 320x240 have been converted to gray scale for the sake of the runtime saving without any considerable loss of accuracy. The conversion is performed by using an approximation of the actual color conversion equation (1).

$$I = .2989R + .5870B + .1140B \ (1)$$

Changing the weights to powers of two for more efficient calculation result in (2).

$$I = .25R + .50G + .25B \ (2)$$

The image is then converted to an AXI4 stream, implemented by Vivado as a communication protocol between the processor, IP cores, and memory. Finally, the AXI4 stream of the image is then converted to a single channel matrix to begin processing.

Pre-processing before the CHT algorithm is performed as discussed. First, the Gaussian 5x5 blur filter is applied using a kernel size of 5x5. The actual value of the edge depends on the magnitude of the gradient. The pixels with relatively high gradient values are considered an edge point. This step is critical for further application of the transform; therefore, the threshold value is a parameter to be adjusted interactively or selected for the application. This value should be chosen carefully on a project basis. If the threshold is too low, many edge points will be found, including duplicates around the actual edge due to the natural image blur. This may be caused by the low contrast surrounding edges or objects of similar colors and intensities. On the other hand, very high threshold values can cause many edge points to be left behind. The edge points are the input values for the CHT; therefore, the more edges present the larger amount of memory to be consumed, adversely affecting the resources and the speed of the algorithm. If the threshold is too high, it will make detecting circles more difficult.

The image goes through the CHT algorithm and an accumulator with all the votes is created for a range of radii. The range of the radii depend on the application and is limited by the amount of memory available described in later chapters. The visualization of the circle is done by looping through the accumulator over a kernel selecting the maximum value across the different radii. If the max value is above the threshold then the points are plotted in the image space creating an artificial circle over the image.

3.2 Gaussian filter

In a real word environment, images are not expected to be of perfect quality. In other words, the images are noisy. Charge-Couple Devices (CCD) imaging is known to have noise of various types (due to the camera heat and the electronic noise). It is modelled generally as an Additive White Gaussian Noise (AWGN). The Gaussian filter enhances the image by convolving the image window with the “bell-shape” Gaussian function that smoothens the noise. This allows the input into the Sobel Edge Detector to have a lower noise impact, and thus spurious edges due to the noise are reduced. Also, a certain amount of blur reduces the chance of edges created by small details, such as texture and small objects, to be taken out of further processing. A kernel size of 5x5 and a $\sigma = .8$ were chosen, as it allowed for the best results from the end-result quality in our experiments.

Given the kernel, the smoothing is performed by convolving the filter with image pixels and producing a 5x5 window of image pixels in which the central pixel value is a sum of products divided by the normalization constant. The row and column margins are not processed. Alternatively, one can process the margins by padding with zeroes or values of neighboring pixels outside the boundaries of the image. For the xfOpenCV library function, the implementation replaces the floating-point arithmetic typically used with fixed point arithmetic using look-up tables (LUT) for multiplication. **Fig 1** shows a comparison of results obtained by the edge detector and thresholding with and without the Gaussian filter.

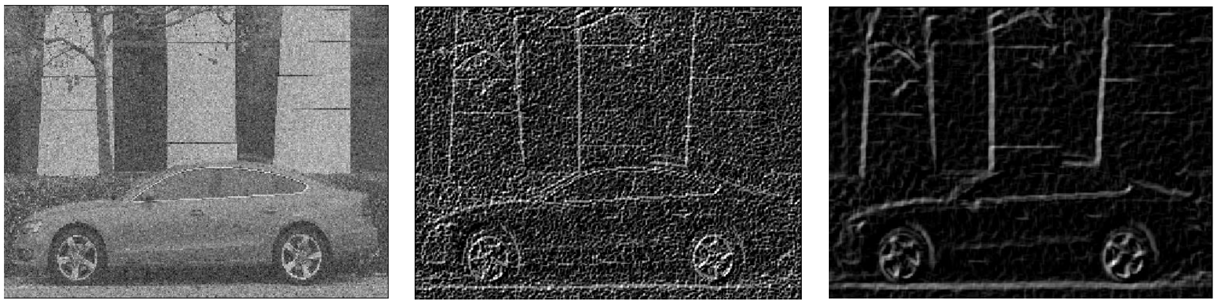


Fig 1. Noisy Image (Left) Edge Detector without filter (Center) Edge Detector with Filter (Right)

3.3 Sobel Edge Detection

Edge detection is a preliminary step in most methods in which the outline of objects is needed. This is especially crucial in the implementation of Hough Transform algorithms. In certain cases, the edge detection may be done during preprocessing in the FPGA Programmable Subsystem (PS) and pipelined to the Programmable Logic (PL) for further processing. This mainly applies to solutions in which memory is more of a concern than the speed. By applying Sobel Edge Detection, it will allow the CHT to find edges of the circle to vote on.

The Vivado HLS 2019 suite provides several functions of OpenCV libraries, which includes xfOpenCV Sobel Edge Detection library function solution that is used in this work. This operation uses a 3x3 mask for calculating vertical and horizontal gradients as the sum of 3 pixels on the left and the right, as well as the difference of sums on the top 3 and the bottom 3 pixels, weighted for normalization.

The convolutions are shown in (1) and (2) in which $\mathbf{G_x}$ and $\mathbf{G_y}$ are the derivatives in the horizontal and vertical directions, respectively, and **A** is the 2-dimensional matrix of intensity values in a 3x3 window of the enhanced Gaussian image.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \qquad (1)$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad (2)$$

At each point in the image, the approximation of the total gradient can be combined normally using (3).

$$G = \sqrt{G_x^2 + G_y^2} \qquad (3)$$

However, taking the square root of an equation would require a lot of hardware and time. A function implemented by xfOpenCV takes a weighted sum in which both derivatives are weighted equally simplifying to (4).

$$G = |G_x| + |G_y| \qquad (4)$$

Fig 2 shows the results of the horizontal gradient (left), the vertical gradient (middle), and the weighted sum of the two (right). One can argue that a more sophisticated edge detector, which eliminates duplicate edges and considers only connected points to report edges, such as Canny edge detector, would result in more significant edge detection. However, for the sake of the acceleration of the overall algorithm, we do not consider it in this work.

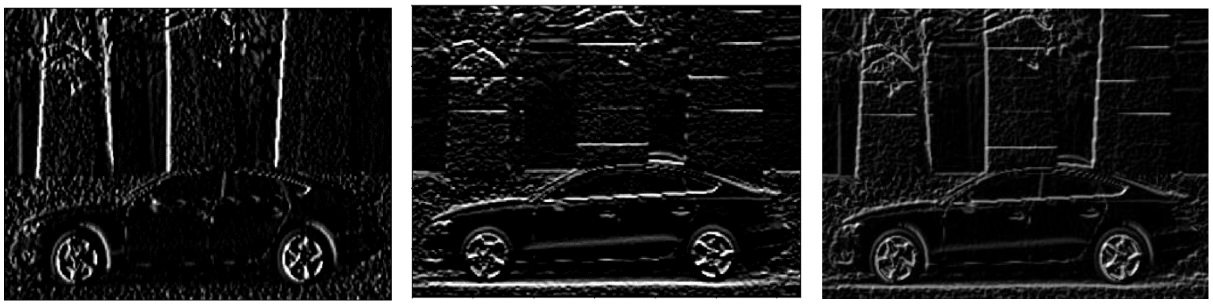


Fig 2. Sobel in the horizontal direction (Left) Sobel in the vertical direction (Center) Sobel summed together (Right)

3.4 Thresholding

Thresholding is performed by Vivado HLS's xfOpenCV library as well. The goal of thresholding is to ensure that only pixels above a certain threshold are selected creating a completely binary image. This also helps eliminate thick or false edges created by the Sobel Edge Detection. Thick edges could be further eliminated by using a morphological filter. The step function shown in (5) describes the thresholding used to discriminate between edge and non-edge pixels.

$$dst(I) = \begin{cases} maxval & if\ src(I) > thresh \\ 0 & otherwise \end{cases} \qquad (5)$$

For visualization, the edge points are displayed as white (value is 255) on a black background. This is a regular image binarization procedure required before the CHT is performed. The threshold needed depends on the image and may need to be adjusted per application. A threshold of 35 is used to result in **Fig 3**, which was found experimentally.

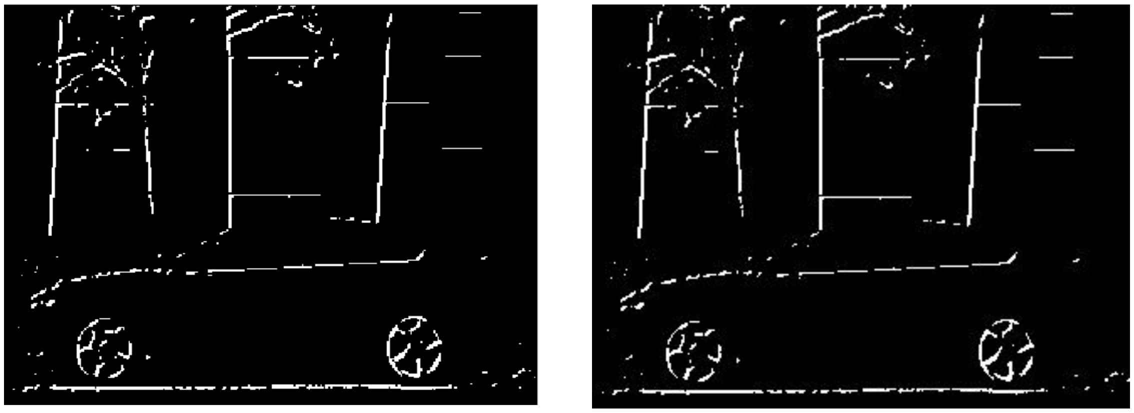


Fig 3. Thresholding in Python (Left) Thresholding in Vivado HLS (Right)

3.5 Circle Hough Transform

The Hough Transform utilizes the contour points resulting from edge detection and uses a voting process to detect patterns of points in binary image data. It was first patented by Paul Hough in 1962 and later suggested by Duda and Hart as a method which uses the polar coordinate representation of the line length, ρ and orientation, θ, of the normal vector to the line from the origin of the image [8]. The length of the line can be derived as shown in (6) in which x and y represent a coordinate pixel on the image.

$$\rho = x \cos(\theta) + y \sin(\theta) \quad (6)$$

These polar coordinates can be visualized in **Fig 4** as a relationship to (x,y).

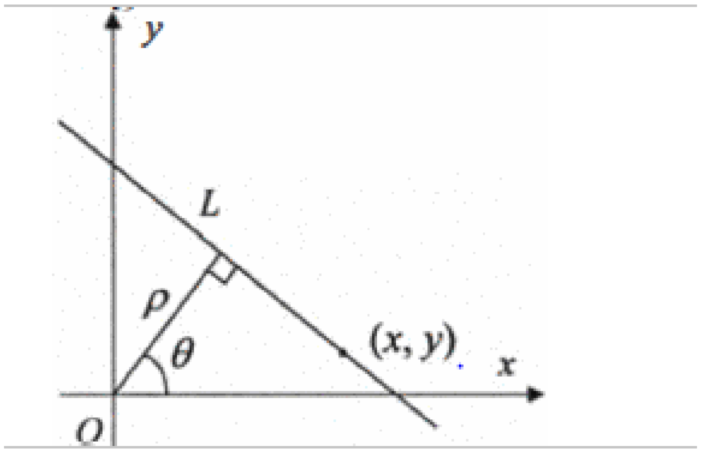


Fig 4. Relationship between (x,y) [11]

The method has then been adapted to detect parabolas and ellipses. For this work, the focus is on detecting circles with the CHT algorithm. The CHT implemented in this work consists of three parts. First, the image is scanned for the edges found after Gaussian smoothing and thresholding for edge detection. Then, circles of varying radii on the detected edges are calculated and accumulated in a bin in the parameter space. Finally, the accumulator array is iterated in windowed sections at a time to locate the local maxima and only draw the circle's maximum value whose center is above the threshold.

The principles for the CHT are as follows. The general equation of a circle is as follows:

$$(x - a)^2 + (b - y)^2 = r^2 \quad (7)$$

Using the basics of trigonometry and a given radius, any point on a circle can be calculated by (8) and (9).

$$a = x - R\cos(t) \quad (8)$$

$$b = y - R\sin(t) \quad (9)$$

Where t is changes from 0 to 360 degrees and (a,b) are any one point around the circle of radius R centered at (x,y). **Fig 5** shows the derivation of (8) and (9) using the unit circle and the Pythagorean Theorem. Knowing the center (x,y) and the radius, any circle can be drawn from those equations.

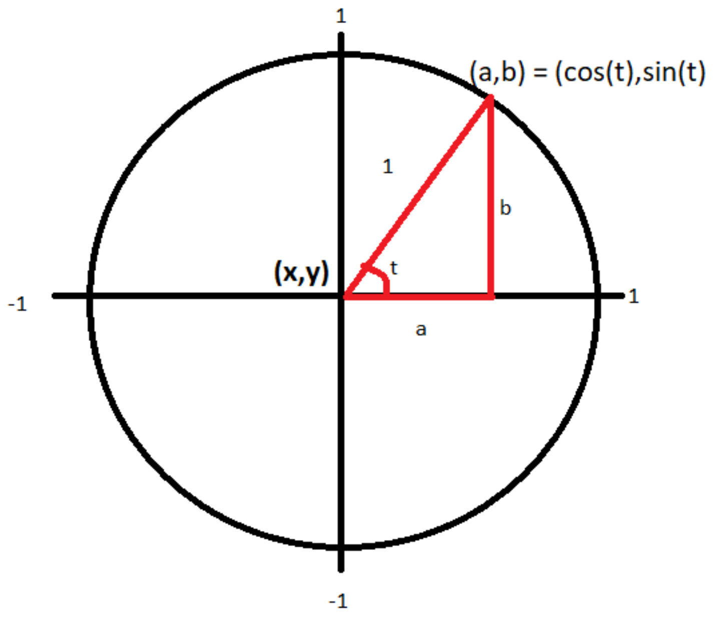


Fig 5. Using Pythagorean Theorem to find a point on a circle

Fig 6 shows points found within the edges of a circle. A blueprint of a circle is created around that point edge and the value of the pixels found are “voted” and incremented in an accumulator array the size of the image. Once the algorithm parses through the entire image, the points with the most votes denote the existence of the center of a circle within the image at a given radius.

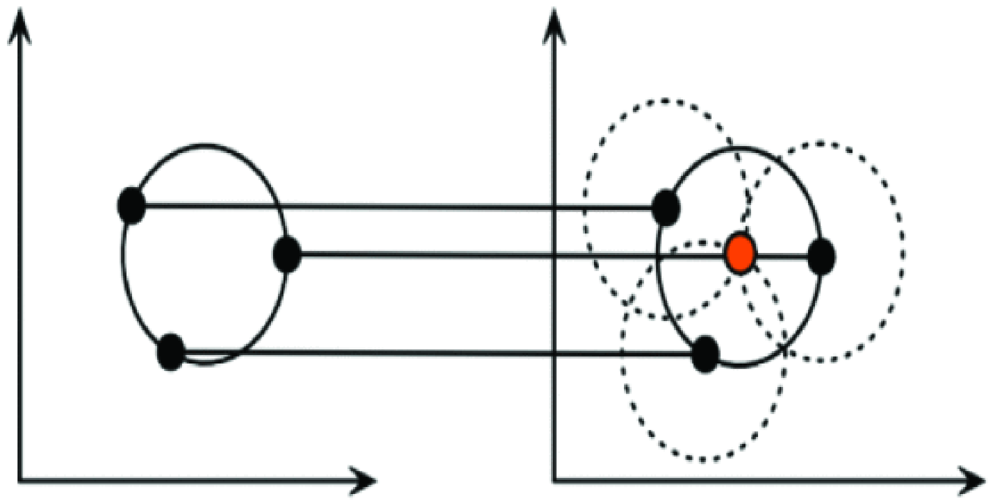


Fig 6. Points in geometric space (left) and points in parametric space (right) [11]

Although there are CORDIC algorithms for calculating many trigonometric functions, the decision is made to calculate sin and cos function by the simple look-up. This allows for reduction of the number of cycles taken by the iterative nature of CORDIC. Instead, two arrays were created for storing the calculations for every sin and cos values from 0 to 360 degrees. Reducing the amount of iterations by using a different incremental value can speed up the algorithm but reduce the accuracy of the circles detected and how they are visually drawn.

In most cases, the actual radius of the circle is not known. This is an additional, but necessary parameter when detecting circles of varied radii which adds complexity to the algorithm. As seen in **Fig 7**, adding an unknown radius adds a third dimension to the accumulator that keeps track of the votes at different radii thereby increasing the memory required. Knowing the radii of the circles in the imagery would greatly decrease the use of memory needed. An extensive search can be implemented without iterating through each radius by changing the increment variable.

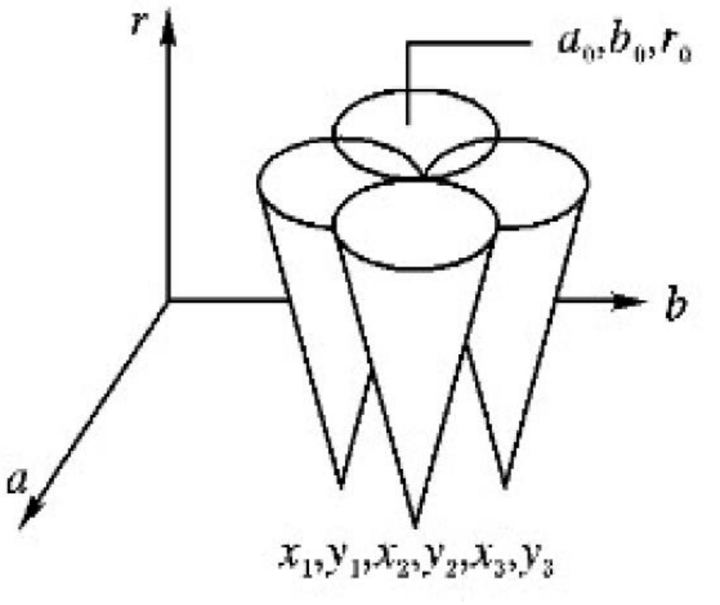


Fig 7. Multiple circles in the parameter space drawing at different radii form a cone [11]

In the CHT voting algorithm, first, a 3D accumulator array the size of the image and of the number of radii must be initialized with zeroes. Then, look-up table arrays for every angle from 0 to 360 degrees of sin and cos are initialized. The CHT algorithm below iterates through every pixel coordinate within the image bounds. If an edge was found, multiple circles of a range of radii are iterated, each incrementing the voting accumulator array.

Algorithm 2.1 Circle Hough Transform Voting Algorithm

```
 1:  Initialize: accum [Rows][Cols][Radius] = 0
     Initialize: sin [] and cos [] loop up table arrays for every angle n from 0 to 360 degrees
 2:  for each x in Row do
 3:    for each y in Cols do
 4:      if cell(x,y) != 0 then //Look for edge
 5:        for each r in Radius do
 6:          for each $n \in (0,360)$ do
 7:            b = y – r * sin[n]
 8:            a = x – r * cos[n]
 9:            if a $\epsilon$ (Rows, Cols) and b $\epsilon$ (Rows, Cols) then
10:              accum[x][y][r] += 1 //Voting
11:            end if
12:          end for
13:        end for
14:      end if
15:    end for
16:  end for
17:  end for
```

After the accumulator array is filled and each radius is accounted for, the algorithm searches and selects the highest vote in the accumulator array within a section of the image. Depending on the performance of the edge detection procedure, many centers can be detected causing false detections. To mitigate this, two methods are utilized. First, a threshold is set to allow only votes higher than that threshold to be counted as centers. However, this could be insufficient because the centers for different radii could still be present. In this implementation of the CHT algorithm, a window is used to traverse through the accumulator array at different radii and select only the local maximum for that window.

Algorithm 2.2 Circle Hough Transform Selection Algorithm

Require: Kernel Size $K > 0$ is the size of the window to search through; Circle Threshold C is the threshold required to consider a vote a pixel. I_Dst(Rows,Cols) is the destination image.

```
 1: Initialize *pixel* which will keep track of the highest vote for the pixel in the
    accumulator array
    Initialize *x0,y0,r0* which will keep track of the index of the highest voted pixel
    Initialize *temp* which will temporarily hold the highest vote
 2: for each x in Row do
 3:   for each y in Cols do
 4:     pixel = 0, temp = 0
 5:     for each i in $K$ do
 6:       for each k in $K$ do
 7:         for each r in Radius do
 8:           *temp* = accum[x+i][y+j][r]
 9:           if *temp* > *pixel* then
10:             p*ixel* = *temp*
11:             x0 = x+i
12:             y0 = y+j
13:             r0 = r
14:           end if
15:         end for
16:       end for
17:       if $pixel > C$ then
18:         for each $n \in (0,360)$ do
19:           b = y0 – r0 * sin[n]
20:           a = x0 – r0 * cos[n]
21:           if a and b $\epsilon$ (Rows, Cols) then
22:             I_Dst(a,b) = 255
23:           end if
24:         end for
25:       end if
26:     end for
27:   end for
28: end for
```

If the maximum value passes the selected threshold, then a circle center is registered in that section of the image. This makes it so that if a “voted” accumulation point did surpass the threshold that only the circle with the most votes in that section is used to visualize the circle in the image. The size of the window would depend on the range of radii. **Fig 8** demonstrates the results of the CHT implementation.

(a)

(b)

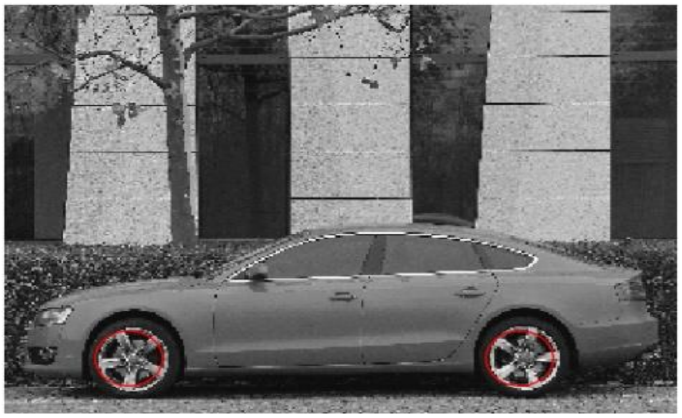

(c)

Fig 8. (a) Original Image (b) HLS implementation of Circle Hough Transform (c) Python implementation of Circle Hough Transform

3.6 HLS Implementation

Reprogrammable FPGA based designs offer the design flexibility and low power implementation of hardware acceleration options compared to ASIC and GPUs. HLS allows the compilation of synthesizing circuits directly from the high-level languages, such as C and C++ codes. Images can be converted to an AXI4 stream which is Vivado's communication protocol between the processor, Intellectual Property (IP) cores, and memory. This includes AXI4-Stream Video which is a subset of the streaming protocol specifically designed for the transportation of video frames. This greatly increases the speed of the development for software and hardware engineers to implement and make changes to IP cores designed for fabric.

Optimization options and tools only available with a hardware implementation can be used to improve the performance of the algorithm that would not be available in Python. For example, PRAGMA compiler directives are used for controlling the type of processing, unrolling the loops, pipelining, partitioning, and reshaping the arrays for attaining parallel and pipelined architectures. This means that some steps inside loops can be performed within the same clock cycle as others thereby reducing the latency. The Gaussian Filter, Sobel Edge Detection, and Thresholding have already been optimized by Xilinx's xfOpenCV library for embedded systems.

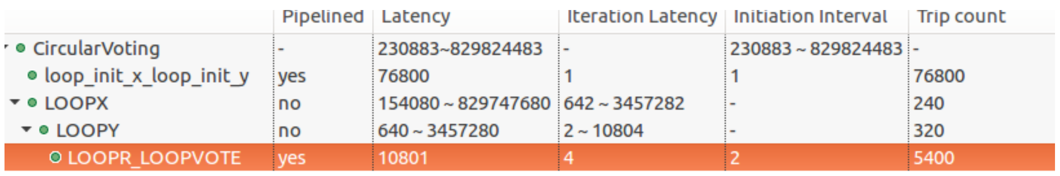
Two functions were implemented for the CHT algorithm that utilize the pipelining tools. First, the CircularVoting function initializes the accumulator array and scans the image of edges at different radii to create an array of “votes”. The array has been completely partitioned in the third dimension, which splits the array into its individual elements for faster access. With the help of PRAGMAS, in **Fig 9** the *LOOPVOTE* is pipelined. This means that the next (a,b) points can be calculated at an earlier time than that of the next circle being calculated. The ability to pipeline is a matter of dependencies. An 8-bit variable called *votes* was created to temporarily store the current vote form BRAM before updating the value since reading and writing the memory at the same time would cause a data hazard due to dependency that would prevent the loop from being pipelined.

```
    LOOPR:for(ap_uint<10> r =R_MIN; r<R_MAX; r+=R_INCREMENT){
#pragma HLS LOOP_TRIPCOUNT min=0 max=100
        row_index = 0;
        LOOPX:for(ap_uint<10> x = 0; x < height; x++ ){
#pragma HLS LOOP_TRIPCOUNT min=1 max=ROWS
            XF_PTNAME(DEPTH) img_pixel_val;
            LOOPY:for(ap_uint<10> y = 0; y < width; y++){
#pragma HLS LOOP_TRIPCOUNT min=1 max=COLS
#pragma HLS DEPENDENCE  array inter false
#pragma HLS LOOP_FLATTEN off
                img_pixel_val = _src_mat.read(row_index); // Reading next pixel
                if(img_pixel_val != 0) //Found Edge
                {
                    LOOPVOTE:for(ap_uint<10> n = 0; n < ANGLEN; n+=ANGLE_INCREMENT ){
#pragma HLS LOOP_TRIPCOUNT min=0 max=360
#pragma HLS PIPELINE
                        b = y - r *  sinval[n];
                        a = x - r * cosval[n];
                        if(a >= 0 && a < height && b >= 0 && b < width){
                            ap_uint<8> r_index = (r-R_MIN)/R_INCREMENT;
                            votes = accum[a][b][r_index];
                            votes = votes + 1;
                            accum[a][b][r_index] = votes;
                        }
                    }//END LOOPVOTE
                }
                row_index = row_index + 1;
            }//END LOOPY
        }//END LOOPX
    }//END LOOPR
```

Fig 9. CircleVoting Function Implementation

The results of pipelining can be analyzed using Vivado's HLS Analysis tool. By inserting a *LOOP_TRIPCOUNT* to for loops inside the code, Vivado can estimate the latency required to complete the loop for variable number of iterations. **Table 1** shows the latency results and the loops that were pipelined for the CircularVoting function. Without pipelining the max latency is almost more than half compared to pipelining. This latency measures the number of clock cycles required for the function to compute all output values. LOOPR and LOOPVOTE have also been flattened together automatically to reduce timing.

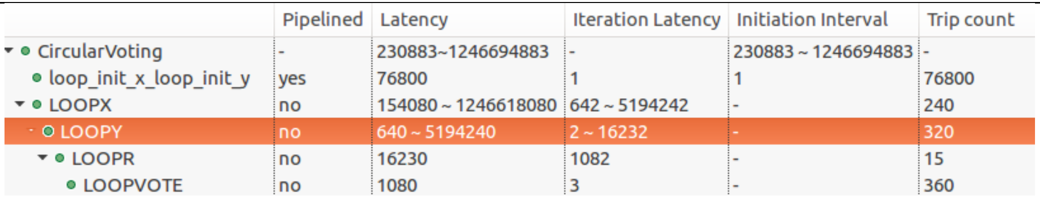
W/ Pipelining



	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
CircularVoting	-	230883~829824483	-	230883 ~ 829824483	-
loop_init_x_loop_init_y	yes	76800	1	1	76800
LOOPX	no	154080 ~ 829747680	642 ~ 3457282	-	240
LOOPY	no	640 ~ 3457280	2 ~ 10804	-	320
LOOPR_LOOPVOTE	yes	10801	4	2	5400

(a)

W/O Pipelining



	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
CircularVoting	-	230883~1246694883	-	230883 ~ 1246694883	-
loop_init_x_loop_init_y	yes	76800	1	1	76800
LOOPX	no	154080 ~ 1246618080	642 ~ 5194242	-	240
LOOPY	no	640 ~ 5194240	2 ~ 16232	-	320
LOOPR	no	16230	1082	-	15
LOOPVOTE	no	1080	3	-	360

(b)

Table 1. Pipelining comparison for CircularVoting

The second function, CircularSorting(), iterates through the array and finds whether a point is above the threshold. For every (x,y) in the image a window searches for the maximum value as it is done in Selection Sort. If there is a center that passes the *CIRCLE_THRESHOLD*, it is drawn.

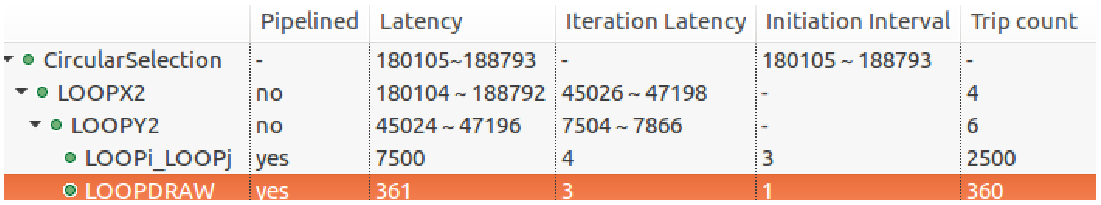
```
 LOOPX2:for(ap_uint<10> x = 0; x < height-KERNEL_SIZE; x+= KERNEL_SIZE){
#pragma HLS LOOP_TRIPCOUNT min=1 max=ROWS
     XF_PTNAME(DEPTH) img_pixel_val;
     LOOPY2:for(ap_uint<10> y = 0; y < width-KERNEL_SIZE; y+= KERNEL_SIZE){
#pragma HLS LOOP_TRIPCOUNT min=1 max=COLS
#pragma HLS LOOP_FLATTEN off
         // Find the local maximum
         pixel = 0;
         temp = 0;
         LOOPi:for(ap_uint<13> i = 0; i < KERNEL_SIZE; i++){
#pragma HLS LOOP_TRIPCOUNT min=1 max=ROWS
             LOOPj:for(ap_uint<13> j=0; j< KERNEL_SIZE; j++){
#pragma HLS LOOP_TRIPCOUNT min=1 max=COLS
#pragma HLS PIPELINE
                 LOOPR:for(ap_uint<13> r=R_MIN; r<R_MAX; r+=R_INCREMENT){
#pragma HLS LOOP_TRIPCOUNT min=1 max=100
                     ap_uint<8> r_test = (r-R_MIN)/R_INCREMENT;
                     ap_uint<10> x_i = x+i;
                     ap_uint<10> y_j = y+j;
                     temp = accum[x_i][y_j][r_test];

                     if(temp > pixel){
                         pixel = temp;
                         x0 = x_i;
                         y0 = y_j;
                         r0 = r;
                     }
                 }//END LOOPR
             }//End LOOPI
         }//END LOOPJ
         if(pixel >= CIRCLE_THRESHOLD){
             LOOPDRAW:for(ap_uint<10> n = 0; n < ANGLEN; n+= ANGLE_INCREMENT
){
#pragma HLS LOOP_TRIPCOUNT min=0 max=360
#pragma HLS PIPELINE
                 b = y0 - r0 * sinval[n];
                 a = x0 - r0 * cosval[n];
                 if(a >= 0 && a < height && b >= 0 && b < width){
                     row_index = a*width + b;
                     _dst_mat.write(row_index, 255);
                 }
             }//END LOOPDRAW
         }
     }//END LOOPY
 }//END LOOPX
```

Fig 10. CircularSorting Function Implementation

Again, the loops searching for the maximum value within the accumulator array and the drawing of the circles in the array have been pipelined.

W/

Pipelining

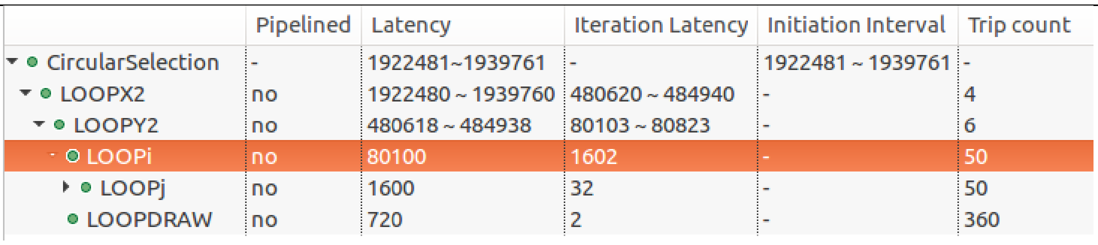


	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
CircularSelection	-	180105~188793	-	180105 ~ 188793	-
LOOPX2	no	180104 ~ 188792	45026 ~ 47198	-	4
LOOPY2	no	45024 ~ 47196	7504 ~ 7866	-	6
LOOPi_LOOPj	yes	7500	4	3	2500
LOOPDRAW	yes	361	3	1	360

(a)

W/O

Pipelining



	Pipelined	Latency	Iteration Latency	Initiation Interval	Trip count
CircularSelection	-	1922481~1939761	-	1922481 ~ 1939761	-
LOOPX2	no	1922480 ~ 1939760	480620 ~ 484940	-	4
LOOPY2	no	480618 ~ 484938	80103 ~ 80823	-	6
LOOPi	no	80100	1602	-	50
LOOPj	no	1600	32	-	50
LOOPDRAW	no	720	2	-	360

(b)

Table 2. Pipelining comparison for CircularSelection

The synthesized circuit can then be packaged as an IP core and exported for the use in the Vivado IDE for integration with other cores and the PS. The C code is converted into a Hardware-Description language (HDL) such as Verilog. The tool allows to generate the HDL design modules.

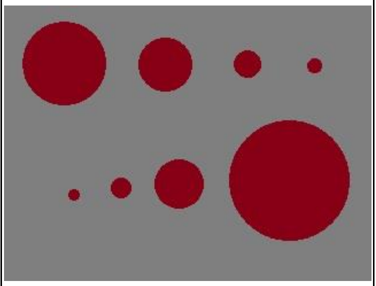
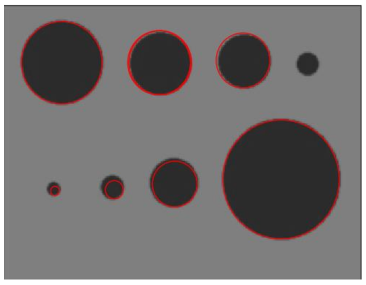
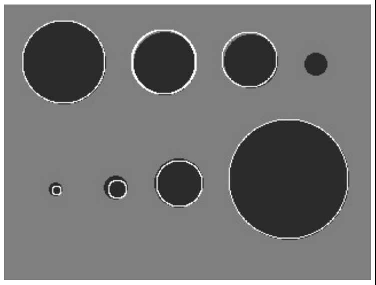
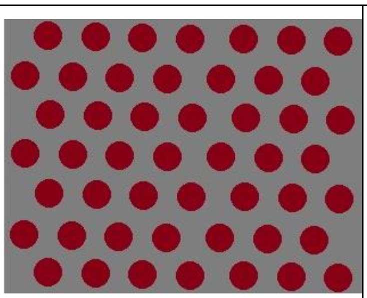
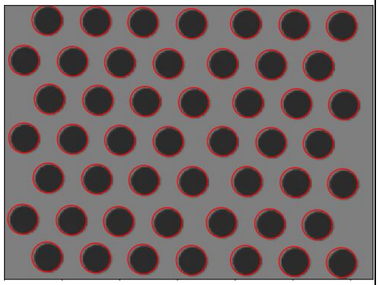
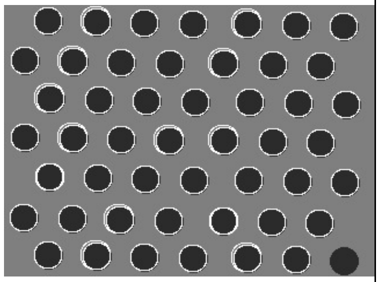
Chapter 4: Experiments and Discussion

4.1 Results

The subjective quality of the outcome is evaluated by observing the results. The objective quality is measured by comparing the circle centers found by software and hardware, respectively. The hardware implementation is produced using the testbench code. The accuracy of the algorithm using the HLS is evaluated by performing the same steps taken using Python OpenCV on a PC and comparing the results. The Python script was created to replicate the algorithm for comparison on a i5-9600k CPU @ 3.70GHz and 16 Gb of Ram on Windows 10.

Multiple images were explored using the Python and the Vivado HLS algorithms. Limitations and results of processing some of the images will be presented. The parameters used for the processing are provided per each solution. *ThreshVal* is the value used for thresholding in the preprocessing step. *MaxKernelSize* refers to the window size used to search for the max votes in the accumulator. *R_Range* is the radius range to search through where [R_Min, R_Max]. *R_increment* describes the iterator incrementation value used. *Circle_Threshold* is the allowed value for a max value from the accumulator array to be considered a circle.

Experiments 1, 2, 3 (**Table 3**) are examples used for searching circles with different ranges of radii. Searching for a wide range of circle sizes has several different limitations. The main issue is the memory requirement. Since HLS has a limited amount of BRAM, only a small search range can be used. If a broad range is used, a larger *R_increment* is needed to reduce the accumulator array size. This, however, can miss circles whose radius lies in between the increments causing inaccuracies in the data, such those in experiment 1 in which a circle was missed. The radius range should be set to an optimal value on an application to application basis as shown in experiment 2 in which the radius range and the kernel are limited to the size of the circles to detect. Moreover, the greater number of radii that need to be searched, the more time the algorithm will take as it must calculate a circle at each edge for every image.

#	**Original**	**Python**	**Vivado HLS**
1	 (a)	 (b)	 (c)
	ThreshVal = 65, MaxKernelSize = 70, R_Range = [4,60], R_increment = 4 Circle_Threshold = 160		
2	 (d)	 (e)	 (f)
	ThreshVal = 65, MaxKernelSize = 10, R_Range = [10,20], R_increment = 1, Circle_Threshold = 180		

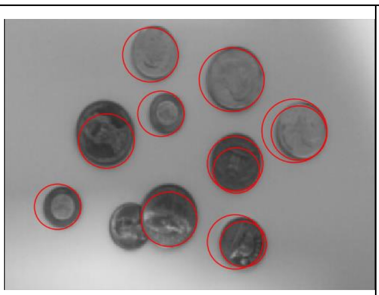
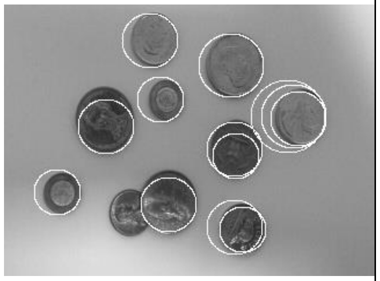
3	(g)	 (h)	 (i)
	ThreshVal = 35, MaxKernelSize = 50, R_Range = [20,80], R_increment = 4 Circle_Threshold = 130		

Table 3. Experiments 1, 2, 3. Images for testing assorted sizes.

Experiment 4 (**Table 4**) is an example of an application of the CHT on iris detection for the eye tracking and iris scanning. This can be used for real time solutions on an embedded platform. However, a limitation arises when trying to locate both the iris and the pupil. Since the pupil sits in the direct center of the eye, its center lies in the same location or approximately close to the same center as the iris. Typically, the larger radius will take precedence over the smaller because more pixel edges will cause a larger vote for that radius within similar areas.

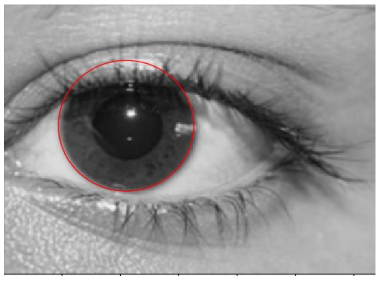
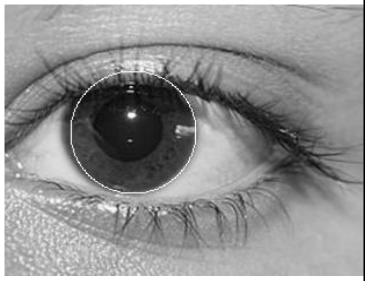
#	**Original**	**Python**	**Vivado HLS**
4	(a)	 (b)	 (c)
	ThreshVal = 30, MaxKernelSize = 50, R_Range = [50,62], R_increment = 4 Circle_Threshold = 210		

Table 4. Experiment 4 Iris Detection

Autonomous driving has become a key point of study in many different fields. The application has strict timing requirements and must be implemented on embedded systems. In Experiments 5, 6, and 7 (**Table 5**) different common detections made for autonomous driving are explored highlighting the importance of the ability to implement a circle detection algorithm on an embedded system. This application can be used for bicycle counting on the roads, road sign detection and detecting of cars (garage or on the streets).

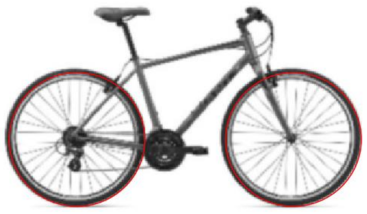
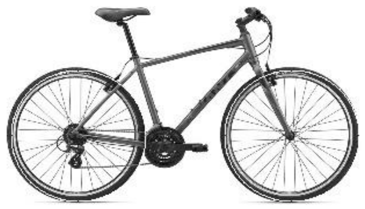
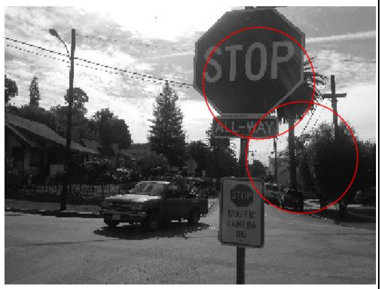
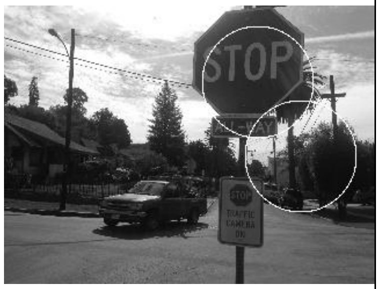
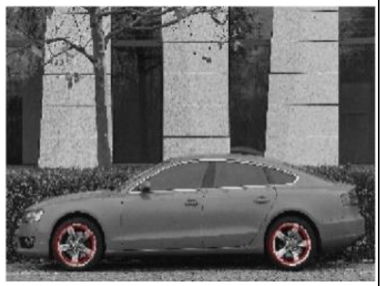
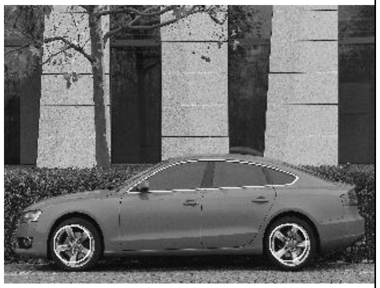
#	Original	Python	Vivado HLS
5	(a)	 (b)	 (c)
	ThreshVal = 85, MaxKernelSize = 100, R_Range = [50,60], R_increment = 2 Circle_Threshold = 130		
6	(d)	 (e)	 (f)
	ThreshVal = 72, MaxKernelSize = 50, R_Range = [48,50], R_increment = 1, Circle_Threshold = 96		
7	(g)	 (h)	 (i)
	ThreshVal = 120, MaxKernelSize = 30, R_Range = [15,20], R_increment = 1, Circle_Threshold = 180		

Table 5. Experiments 5, 6, 7. Autonomous Driving applications.

4.2 Results Performance

To evaluate the performance of the results the following detection metrics were used: True Positive (TP), an object is present in the ground truth (GT) and the experiment; False Positive (FP), an object was present in the experiment but not in the GT; and False Negative (FN), an object is present in the GT, but not in the experiment. Using (12), the Detection Rate (DR), which describes the number of true positives relative to the sum of the true positives and the false negatives, can be calculated to measure the percentage of true targets detected [1].

$$DR = \frac{FP}{(FP+TN)} \quad (12)$$

The results for each experiment are shown in the table below.

	Python				**Vivado HLS**			
#	**TP**	**FN**	**FP**	**DR**	**TP**	**FN**	**FP**	**DR**
1	0.875	0.125	0	87.5%	0.875	0.125	0	87.5%
2	1	0	0	100.0%	0.982143	0.017857	0.214286	98.2%
3	0.9	0.1	0.3	90.0%	0.9	0.1	0.4	90.0%
4	1	0	0	100.0%	1	0	0	100.0%
5	1	0	0	100.0%	1	0	0	100.0%
6	1	0	1	100.0%	1	0	1	100.0%
7	1	0	0	100.0%	1	0	0	100.0%

Table 6. Detection Metric Results

Overall, the results demonstrate a high DR for both the Vivado HLS and Python implementations when each is compared to the original image with an average of 96.5% and 96.7%, respectively. This means that the CHT implementation can be used as an efficient tool for the detection of circles. The algorithm did not perform as well when there was a large range of radii to search for, such as in the third experiment, and when many edges in the background caused inaccurate false positives, illustrated in experiment 6.

One of the main reasons that the two solutions would yield different results is that the approximations at different steps of implementations made to account for speed and memory are affected when targeting an embedded platform. If detection could have been performed after application of more sophisticated algorithms for edge detection, such as in Canny Edge Detection, one could expect a better result. The Vivado HLS resultant images drew less than perfect (more pixelated) circles compared to the Python solution since it used a smaller range of angles to visualize them. The effects of this can be seen when taking the absolute difference between the two images (**Fig 11**).

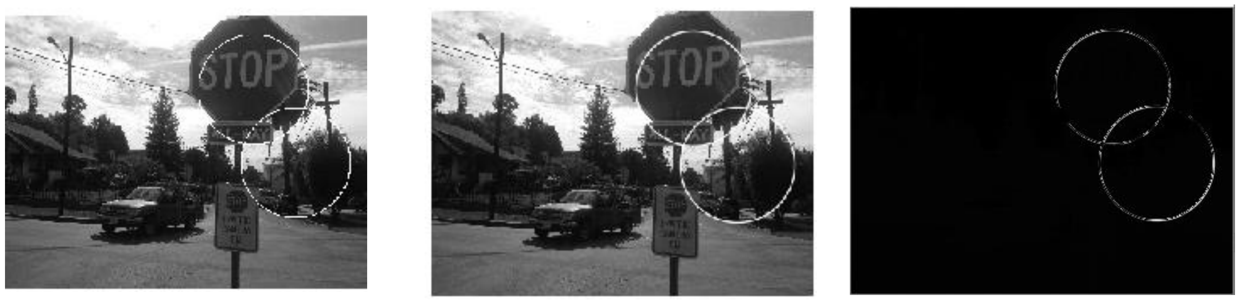
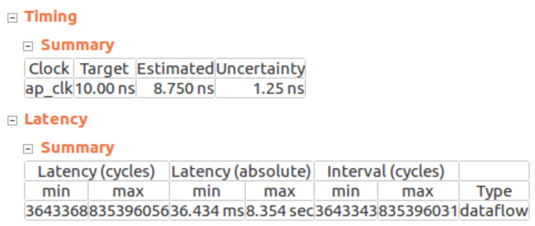


Fig 11. HLS Result (Left) Python Result (Center) Absolute Difference (Right)

4.3 Timing Analysis

Timing of the algorithm varies greatly on the amount of edges detected and the radius count. For each pixel in an edge, the CHT algorithm must find a circle for each radius in the radius range. Vivado provides an estimate of the latency given the parameters described in this work for the run time of the algorithm on the targeted system (UltraScale+ ZCU106 board) at 100MHz. This report is shown in **Fig 12** for experiment 1 in **Table 4Error! Reference source not found.**



Timing

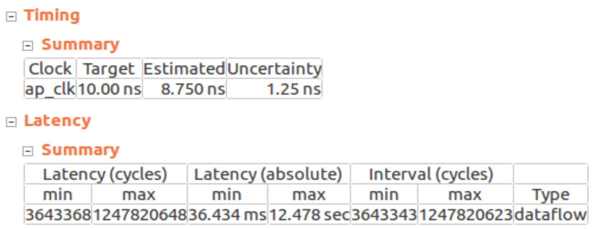
Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.750 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
3643368	835396056	36.434 ms	8.354 sec	3643343	835396031	dataflow



Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.750 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
3643368	1247820648	36.434 ms	12.478 sec	3643343	1247820623	dataflow

Fig 12. Timing and Latency report from Vivado HLS. Pipelined (Left) Not Pipelined (Right)

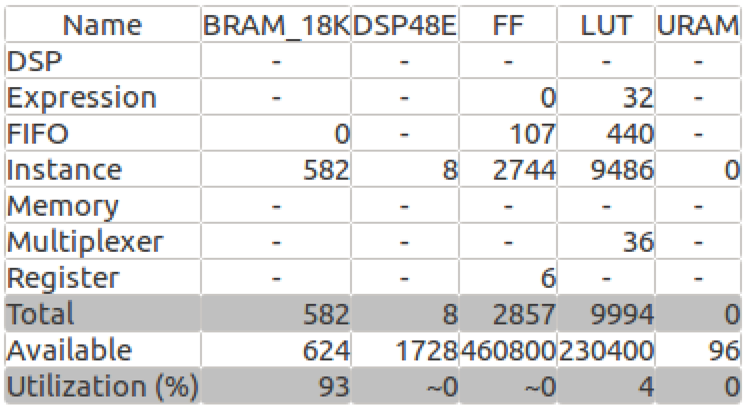
Comparison between the two platforms for experiment 3 is shown in **Table 7**. As tabulated, the minimum and maximum values are the same for the Python PC implementation referred to in the beginning of the results as we can measure the time elapsed between the entire process. However, Vivado HLS only provides an estimate of the time it would take. The results show a clear difference even in the worst-case scenario between a software implementation versus an embedded one. Timing would vary on the loops required to iterate through the different radii.

Platform	Time Analysis min (seconds)	Time Analysis max (seconds)
Vivado HLS	.0364	8.354
Python	5.69	86.71
Difference (%)	99.36	90.37

Table 7. Platform Timing Analysis

4.4 Memory analysis

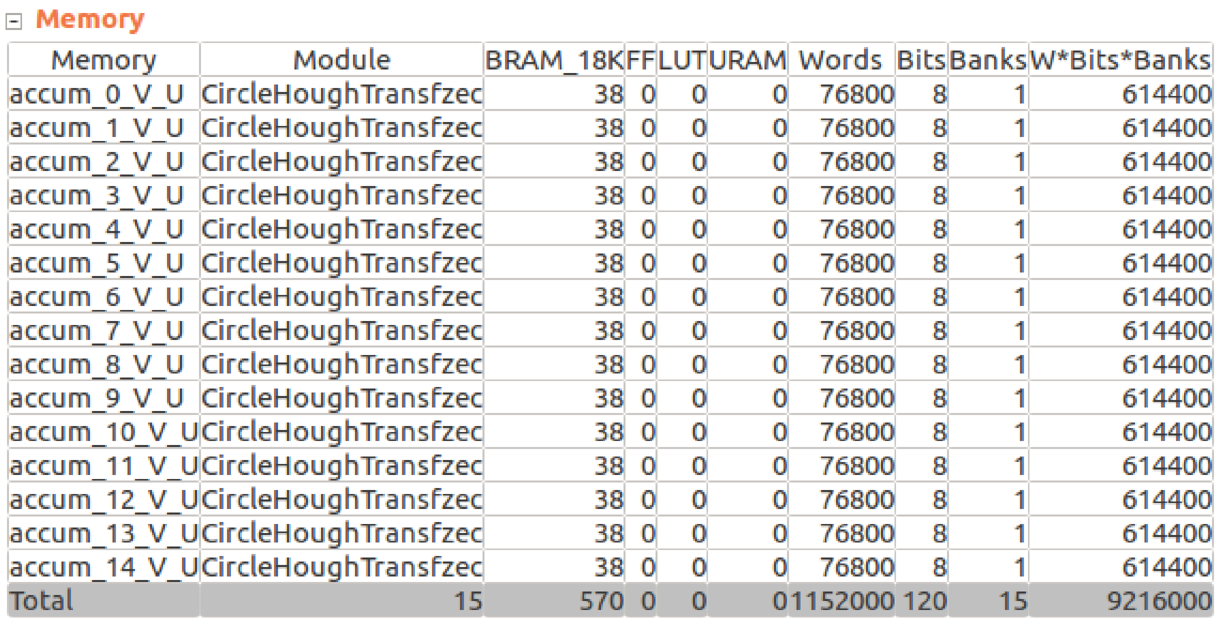
The biggest issue with the Hough Transform is the large accumulation of votes for each pixel and consequently the large memory. The design shows 582 of the BRAM_18k is mainly used to handle the voting array, 8 DSP48E for arithmetic operations, 2857 Flip Flops (FF), and 9994 Look Up Tables (LUT) in **Fig 13** for experiment 3.



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	32	-
FIFO	0	-	107	440	-
Instance	582	8	2744	9486	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	-	-	6	-	-
Total	582	8	2857	9994	0
Available	624	1728	460800	230400	96
Utilization (%)	93	~0	~0	4	0

Fig 13. Memory usage of entire application provided by Vivado HLS

Large image sizes and high amount of edges increase the BRAM usage, which is why a relatively low image size and a relatively high threshold value is used for detection. Therefore, only 8-bit banks are needed. **Fig 14** shows an experiment that 15 instances were created of 38 BRAM_18K. In this case the array is an 8-bit array of 320x240 which is 76800 words and a radius of 15. The reading and writing of the BRAM are implemented using a true dual-port RAM with the control for both read and write on both ports. This allows for a smaller number of cycles for reading the memory but also increases the amount of BRAM needed.

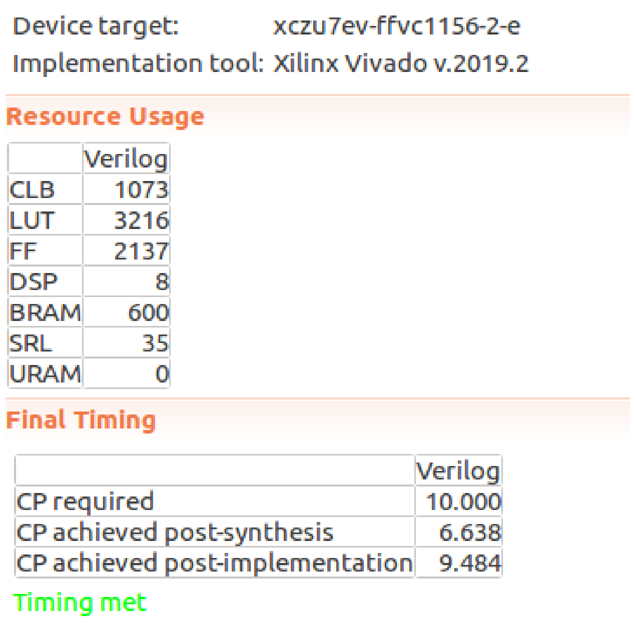


Memory

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
accum_0_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_1_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_2_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_3_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_4_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_5_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_6_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_7_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_8_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_9_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_10_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_11_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_12_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_13_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
accum_14_V_U	CircleHoughTransfzec	38	0	0	0	76800	8	1	614400
Total	15	570	0	0	0	1152000	120	15	9216000

Fig 14. Memory usage of entire application provided by Vivado HLS

After Synthesis, the project can be exported, and the real timing and routing of the design can be measured. **Fig 15** shows the actual resources used required for implementation if placed in a Vivado project. The results tend to use more BRAM but optimizes the resource allocation in FFs and LUTs.



Device target:	xczu7ev-ffvc1156-2-e
Implementation tool:	Xilinx Vivado v.2019.2

Resource Usage

	Verilog
CLB	1073
LUT	3216
FF	2137
DSP	8
BRAM	600
SRL	35
URAM	0

Final Timing

	Verilog
CP required	10.000
CP achieved post-synthesis	6.638
CP achieved post-implementation	9.484

Timing met

Fig 15. Export output usage for targeted device IP core

Chapter 5: Conclusion

5.1 Limitations

When working on embedded platforms there will always be a tradeoff between memory, speed, and the accuracy. As discussed in the experiments, there are several limitations of implementing the Circle Hough Transform. The most critical issue is a large amount of memory needed for data storage. When searching a broad range of radii, the need for BRAM greatly increases. If larger images are needed, the size of the BRAM could be a bottleneck and the solution is to use the external memory (SDRAM), which will increase the latency. If images have too many edges, the voting array implementation is limited to an 8-bit counter. Many scenarios need to be considered when using this implementation to a specific application. However, the design turns out to be highly advantageous where speed is a concern.

5.2 Recommendation

To improve the algorithm, a more efficient edge detection algorithm would need to be implemented into the HLS solution. Algorithms such as Harris or Canny Edge detection could provide more edges to count during the accumulation process. **Fig 16** is an experimental result using one test that could be used for autonomous driving detection using Canny in Python instead of Sobel Edge detection.

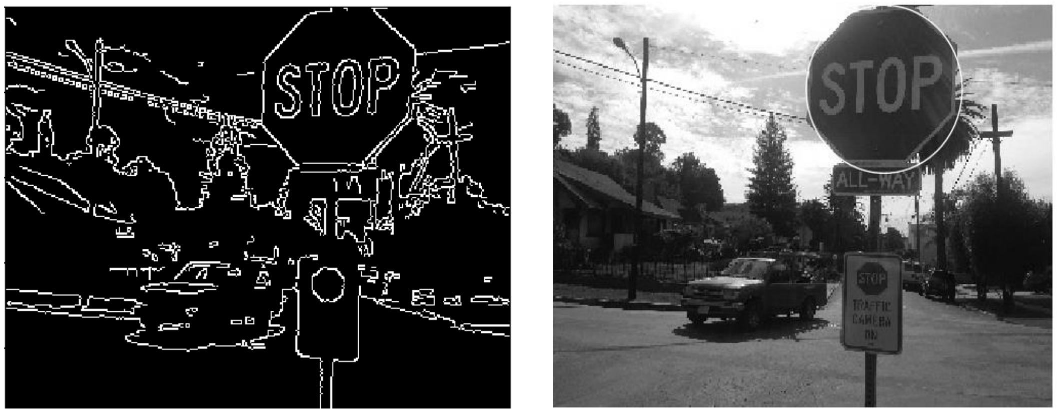


Fig 16. Python Canny Edge Recommendation

After some improvements have been made, the future of this work would be to implement the exported HDL core into an embedded system and test it using either pre-loaded images or real-time video from a camera. Furthermore, not every range of radii can be searched and is limited by the size of the image being processed. It would be most beneficial to know the exact radius being searched by preprocessing the image and predetermining the parameters for the best performance.

The Gaussian Blur has been optimized by the xfOpenCv but could also be further improved by implementing shifts. The sum of products can be implemented by MAC (multiply accumulate) or calculated in parallel using 24 full adders. This can be further optimized using the hierarchy of Carry-save adders. The actual kernel weights are approximated using powers of two instead of the floating-point numbers in the actual 5x5 Gaussian kernel below.

0.003765	0.015019	0.023792	0.015019	0.003765
0.015019	0.059912	0.094907	0.059912	0.015019
0.023792	0.094907	0.150342	0.094907	0.023792
0.015019	0.059912	0.094907	0.059912	0.015019
0.003765	0.015019	0.023792	0.015019	0.003765

Table 8. Actual Gaussian Filter Kernel

	1	1	2	1	1
	1	4	8	4	1
1/84 x	2	8	16	8	2
	1	4	8	4	1
	1	1	2	1	1

Table 9. Integer approximation of Gaussian Filter for the circuit implementation

The result is divided once by the normalization constant which is the sum of all weights in 5x5. This is 84 to keep the output range within the actual range of pixel values and is an integer division which can be implemented by a parallel array divider (no remainder).

5.3 Conclusions

In this work we have investigated and designed the circuit for the implementation of Circle Hough Transform. The design has been performed in Vivado’s High Level Synthesis targeting the UltraScale+ ZCU106 board. The analysis demonstrates a significant speed up factor of 85.4% using pipelining, which is proceeded by loop unrolling and array partitioning. While this solution sacrifices some accuracy compared to its Python implementation, it performs at the accelerated speeds. The accelerator can be used as a part or a stand-alone product (IP) for detecting circles for a variety of applications, wherein this type of processing is an end product or an intermediate step for the analysis including iris detection for scanning and subsequent analysis, detecting bicycles and cars on the roads, and many more other exciting tasks.

References

1. A. Godil, W. Shackleford, M. Shneier, R. Bostelman, and T. Hong, “Performance Metrics for Evaluating Object and Human Detection and Tracking Systems,” 2014.
2. A. Lopez-Martinez and F. J. Cuevas, “Automatic circle detection on images using the Teaching Learning Based Optimization algorithm and gradient analysis,” *Applied Intelligence*, vol. 49, no. 5, pp. 2001–2016, 2018.
3. A. O. Djekoune, K. Messaoudi, and K. Amara, “Incremental circle hough transform: An improved method for circle detection,” *Optik*, vol. 133, pp. 17–31, 2017.
4. A. Tarik, M. Boussaid, A. Karim, and A. Abdelah, “Improving Road Signs Detection performance by Combining the Features of Hough Transform and Texture,” *International Journal of Computer Applications*, vol. 73, no. 9, pp. 5–9, 2013.
5. F. Ferhat-taleb Alim, K. Messaoudi, S. Seddiki and O. Kerdjidj, "Modified circular Hough transform using FPGA," *2012 24th International Conference on Microelectronics (ICM)*, Algiers, 2012, pp. 1-4, doi: 10.1109/ICM.2012.6471412.
6. J. Avey, “An FPGA-based hardware accelerator for iris segmentation.”
7. L. Wang and L.-Q. Chen, “Concentric circle detection based on chord midpoint Hough transform,” *Journal of Computer Applications*, vol. 29, no. 7, pp. 1937–1939, 2009.
8. R. O. Duda, “Use of the Hough transformation to detect lines and curves in pictures,” *Communications of the ACM*, 01-Jan-1972.
9. S.-C. Pei and J.-H. Horng, “Circle arc detection based on Hough transform,” *Pattern Recognition Letters*, 15-Jul-2002.
10. V. Kumar, A. Asati, and A. Gupta, “Hardware Accelerators for Iris Localization,” *Journal of Signal Processing Systems*, vol. 90, no. 4, pp. 655–671, 2017.
11. V. Kumar, A. Asati, and A. Gupta, “Memory-efficient architecture of circle Hough transform and its FPGA implementation for iris localisation,” *Memory-efficient architecture of circle Hough transform and its FPGA implementation for iris localisation - IET Journals & Magazine*, 2018.
12. X. Chen, L. Lu and Y. Gao, "A new concentric circle detection method based on Hough transform," *2012 7th International Conference on Computer Science & Education (ICCSE)*, Melbourne, VIC, 2012, pp. 753-758, doi: 10.1109/ICCSE.2012.6295182.

Curriculum Vitae

Graduate College,
University of Nevada, Las Vegas

Carlos Lemus

Email: carlosslemus@yahoo.com

Degrees:

Bachelor of Computer Engineering, 2018

University of Nevada, Las Vegas

Dissertation Title: A Circle Hough Transform Implementation using High Level Synthesis

Dissertation Examination Committee:

Chairperson, Dr. Emma Regentova, Ph.D.

Committee Member, Dr. Venkatesan Muthukumar, Ph.D.

Committee Member, Dr. Mei Yang, Ph.D.

Graduate Faculty Representative, Dr. Yoohwan Kim, Ph.D.