2017

Real-time ellipse detection on an embedded reconfigurable system-on-chip

Daniel Roggow
Iowa State University

Real-time ellipse detection on an embedded reconfigurable system-on-chip

by

Daniel Roggow

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor
Phillip H. Jones
Namrata Vaswani

The student author and the program of study committee are solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2017

DEDICATION

To my family, for your love and support.

To my teachers, for your labors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank John Deere for contributing hardware and technical support for this project. Additionally, the expertise and contributions of Tarik Loukili, Shayne Rich, Jesse Bialas, Travis Davis, and Michael Kean were invaluable to this work, producing many fruitful discussions before the final result materialized.

I would also like to thank my coworkers at Iowa State University who contributed to this work. Ben Williams and Quinn Murphy, for your work on the Visualizer, Murad Qasaimeh and Dr. Phillip Jones, for your work on the adaptive thresholding core, and Dr. Joseph Zambreno for your guidance and dedication to the project, often keeping me focused until the wee hours of the morning (the late-night pizza helped).

ABSTRACT

Computer vision algorithms have historically been difficult to deploy in resource-constrained embedded systems. Ellipse detection or fitting is an important subproblem in computer vision, and these algorithms are computationally complex enough to pose significant design challenges when targeting an embedded system problem domain. This work describes a least squares ellipse fitting system targeting the Xilinx Zynq 7000 series of SoCs, and uses a well-known methodology to accelerate our algorithm designed to locate six circular markers in a plane from 0.0930 frames per second (FPS) using a Matlab implementation, to 64 FPS. Additionally, the Zynq implementation also achieves a speed-up of 1.14× over an optimized Matlab implementation running on a conventional workstation. Our results demonstrate the effectiveness of a hardware/software co-design approach for obtaining real-time performance for ellipse detection algorithms in an embedded context. To the best of our knowledge, this work is the first to demonstrate an embedded ellipse detection system capable of processing HD resolution images (1920×1080) at a rate exceeding 60 FPS.

CHAPTER 1. INTRODUCTION

Ellipse detection is an important class of computer vision algorithms because an ellipse is the 2-D projection in an image of a circle from a 3-D scene. Circular and elliptical shapes are often present in both nature and constructed environments, such as those found in a factory or a city, and computer vision systems can use information about the circles or ellipses in a given scene to perform some useful task. Examples of ellipse detection applications can be found in a wide variety of domains such as person detection [1], human feature recognition [2], [3], and object tracking [4], agriculture [5], industrial processing [6], biomedical [7]–[11], oceanography [12], photogrammetry and remote sensing [13], astronomy [14], engineering [15]–[18], automotive [19]–[21] and aerial vehicle [22]. The challenge for designing complete ellipse detection solutions for these domains is twofold. Typically, ellipse detection solutions in these areas do not have the amount of computing resources present in a typical workstation, due to severe limitations on system size, weight, and power consumption. Additionally, the results from processing an input frame must be computed and any system actuation must occur before the next frame arrives from the camera. These are significant challenges, since most ellipse detection algorithms require some sort of low-level input image preprocessing in order to extract the appropriate details on which to operate (e.g., edges). These low-level processes usually operate on all the pixels in an image, and sometimes groups of pixels. This means many computations are necessary to complete the preprocessing in order to prepare the data for the ellipse detection algorithm itself, which also is usually computationally intensive. Therefore, it would appear that ellipse detection algorithms are too computationally complex to process frames at the camera frame rate on typical processors in embedded computer vision systems, but, as will be shown in this work, a hardware/software system architecture can overcome these hurdles and facilitate embedded ellipse detection algorithms.

Ellipse detection algorithms can be classified by method into two main types: clustering/voting and optimization. Clustering/voting methods include the Hough Transform and its varieties [23]–[28], RANSAC [29]–[31], fuzzy clustering [32]–[34], and Kalman filtering [35]. Optimization methods include least squares model fitting [36]–[44], genetic algorithms [7], [45], and maximum likelihood [46]. Clustering/voting methods are able to detect more than one ellipse at a time and are robust to some noise in the input data, but are challenging for embedded systems due to many computations involving trigonometric functions and square roots, significant amounts of memory accesses, and some require iterations to produce a result. Although potentially less memory-intensive and slightly less computationally complex, solutions to the optimization methods are in general iterative, weak to non-Gaussian noise in the data, can only fit a single ellipse at a time, and some techniques contain bias in the fitted ellipse. Possibly due to the complexity of designing a full real-time embedded ellipse detection system, few papers discuss such implementations, but much work exists on improving the performance of particular ellipse detection methods by utilizing FPGAs, ASICs, CAMs, and SoCs.

In this thesis, we present an embedded ellipse detection system to track a target containing six circles in real-time. We use a Xilinx Zynq SoC to accelerate the computationally-expensive portions of our algorithm in the FPGA logic, utilizing a pixel-streaming architecture to apply the standard image processing techniques of grayscale conversion, region-of-interest (ROI) masking, and thresholding. Once the processed frame reaches our application software running on the ARM CPU, contours are extracted and ellipses are detected and fit using a least squares method from the OpenCV library. In this way, we avoid the complexities and challenges associated with implementing the actual ellipse detection ourselves, instead relying on an accepted industry solution. Our design achieves a frame rate of 64 FPS when at least six ellipses are detected in a frame and is 1.14× faster than an optimized Matlab implementation. The system was first tested on an Avnet ZedBoard and then deployed on a Zynq-based camera system. Our system was successfully utilized in a controlled road construction environment, and, to the best of our knowledge, this thesis is the first work demonstrating an embedded ellipse detection system capable of processing HD resolution (1920×1080) images at the camera frame rate.

The rest of this thesis is organized as follows. Chapter 2 discusses the basic mathematical theories behind ellipse detection and briefly describes some of the methods for detecting ellipses found in the literature. Chapter 3 describes several ellipse detection systems and evaluates their suitability for our problem. Chapter 4 introduces our ellipse detection algorithm. Chapter 5 describes our design process and the preliminary performance results guiding our design choices. Chapter 6 details the system architecture, including both software and hardware components. Chapter 7 provides our experimental results and Chapter 8 offers our Conclusion and outlines future work.

CHAPTER 2. BACKGROUND: ELLIPSE DETECTION

An ellipse is one of the three types of conic sections and is the result of intersecting a plane with a cone. The general Cartesian form of a conic section is

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \tag{2.1}$$

and in order for a conic to be an ellipse, it must satisfy the constraint

$$B^2 - 4AC < 0 \tag{2.2}$$

Figure 2.1 depicts some of the main geometrical features of an ellipse. An ellipse (in Cartesian coordinates) is the set of points such that, for some point $P = (p_x, p_y)$ and foci $f_1 = (f_{1x}, f_{1y})$ and $f_2 = (f_{2x}, f_{2y})$,

$$\sqrt{(f_{1x} - p_x)^2 + (f_{1y} - p_y)^2} + \sqrt{(f_{2x} - p_x)^2 + (f_{2y} - p_y)^2} = 2a$$

That is, the sum of the distance from the point to both foci is the length of the *major axis*. The line the foci are on is the *principal axis*. Where the principal axis intersects the conic are *vertices*, and the major axis of the ellipse is the line joining these vertices. The *center*, c, of the ellipse is the midpoint of the major axis, and the foci are equidistant from this point, where

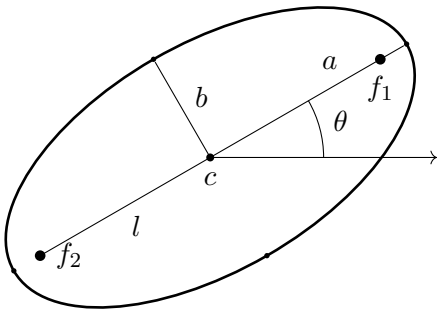


Figure 2.1: Major features of an ellipse.

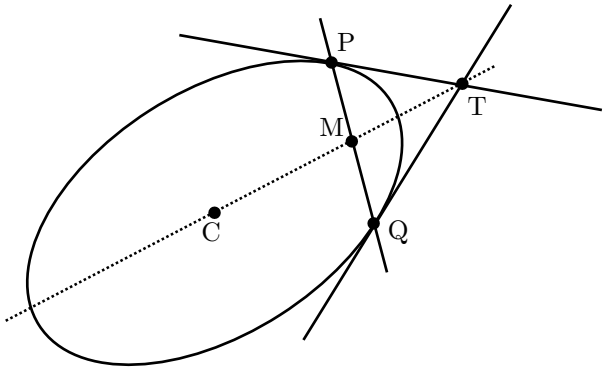


Figure 2.2: The chord-tangent method.

the distance to one of the foci is called the *linear eccentricity* of the ellipse, labeled l in the figure. The *minor axis* of the ellipse is perpendicular to the major axis, and through the center (b in the figure). Conics also have an *eccentricity parameter*, e, and for an ellipse $0 < e < 1$, and can be taken to mean how far the ellipse deviates from a circle. e is a ratio of the distance between the foci to the length of the major axis: $\frac{2l}{2a} = \frac{l}{a}$. The *orientation* of an ellipse can be defined as the angle between the horizontal axis in the plane and the major axis of the ellipse, as denoted by θ in the figure.

As can be seen in Figure 2.1, conics have five degrees of freedom describing their vertical position, horizontal position, orientation, shape, and scale. This means a conic can be uniquely determined by any five points.

An important property of ellipses which is often used in ellipse detection is shown in Figure 2.2. A line drawn through the intersection, T, of the tangents of two points P and Q, and the midpoint, M, of a line connecting P and Q, will also pass through the center of the ellipse (C).

Ellipse detection methods use these properties in various ways to extract information about ellipses in images and can broadly be classified into two main methods: clustering/voting, and optimization, and within these methods are many variations.

2.1 Clustering/voting

This section describes some of the common clustering/voting techniques, which include the Hough Transform and its variants, RANSAC, fuzzy clustering, and Kalman filtering.

2.1.1 The Hough Transform

The Hough Transform (HT) facilitates detection of parameterizable shapes in images by converting information in the input image into a parameter space through a voting process and then extracting the best candidates through a peak-finding process. In its traditional form, the Standard Hough Transform (SHT) uses the line parameterization $\rho = x\cos\theta + y\sin\theta$ and maps $\{\rho,\theta\}$ values in parameter space. Points in image space become lines in parameter space, and lines in image space become points in parameter space.

Typically, HT algorithms involve some sort of image preprocessing to extract or highlight features in the image, such as edges, for the HT computations. These features are used to generate the proper bin to place a vote, which are then collected in an accumulator. After voting is finished, the accumulator is processed so that the appropriate shapes are identified, usually by locating peaks in parameter space. Other processing may occur at this stage to help alleviate spurious identification of shapes.

It can be seen directly that the dimensionality of the parameter space increases with each additional parameter. For instance, using the standard HT approach to detect circles requires a 3-D parameter space, and detecting ellipses requires a 5-D parameter space. Because the computational and storage requirements are very costly for any parameterized shapes besides lines, many variants of the HT have been conceived. In the following discussion, only some of the variants for ellipse detection are considered, but for the interested reader, surveys of various HT methods can be found in [47]–[49].

2.1.1.1 The Generalized Hough Transform

The Generalized Hough Transform (GHT) [24] improves upon the SHT by providing a mechanism for arbitrary non-analytic shapes to be identified in addition to parameterizable

shapes, even under scalings and rotations. The basic premise is to construct a look-up table during algorithm development by selecting a reference point in the shape to act as the origin for a local polar coordinate system, and then constructing a table indexed by gradients at each boundary point on the shape, where each entry in the table contains the radial distance from each point with the given gradient to the reference point, and the angle between each point with the given gradient to the reference point. When the desired shape is at fixed scale and orientation, the parameter space only requires two dimensions, however, when accounting for scale and rotation, the parameter space becomes four dimensional. The algorithm also requires a gradient computation for each detected edge pixel.

2.1.1.2 The Randomized Hough Transform

The Randomized Hough Transform (RHT) [50] seeks to speed up the voting stage of the HT by randomly selecting the minimum number of points from the set of features and computing the shape parameters from those points. If the computed parameters are within some tolerance value of an existing set, the vote for those parameters is incremented, otherwise, the new parameters are stored and the vote initialized to one. According to [50], the RHT cannot be used directly for curves defined by non-linear equations, such as an ellipse, because solving a system of five non-linear equations would require significant computation time, rendering the algorithm useless. [27] proposes a solution for detecting ellipses using the RHT by selecting three points from the set of features (edge pixels, in this case), and estimating the tangents of these three points. The center of the ellipse is then estimated using the method proposed by [51], exploiting the chord-tangent method in Figure 2.2. The three remaining parameters are computed by solving a linear system of three equations. Once computed, the parameters are added to a list of valid parameters if the constraint in Equation 2.2 is satisfied. This variant of the HT for ellipse detection is weak when there are more than two ellipses in an image, or if there is noise in the image, and still remains computationally intensive.

2.1.1.3 Straight Line Hough Transform

[25] proposes a modification to the SHT where ellipses are detected by decomposing the HT into seven stages. The first is similar to the SHT, but forms a linked list based on the computed θ value of edge pixels. Next, the set of possible lines passing through the center of the ellipse is generated by using the principle of diameter bisection for ellipses and computing the the midpoints of all parameter point pairs with the same θ value. Then, the (x, y) intersections of these lines are computed and each intersection casts a vote in a 2-D accumulator bit array. This array is then searched for clusters, and the first guess for the center of an ellipse is determined by the location in a cluster where the sum of "on" bits in a $w \times w$ window centered at that location is the greatest. In order to better estimate the center, the set K of feature points in the input image is collected by computing the distance from the guessed center to the closest feature point at every value of θ, $0 < \theta < 2\pi$. The maximum value $k_1 \in K$ is noted and is assumed to be an endpoint of the major axis for the ellipse. Once all closest points have been found, the farthest point $k_2 \in K$ from k_1 is located, and the true center of the ellipse is taken to be the midpoint of $k_1 k_2$. The orientation can then be computed and the semiminor axis located. Lastly, a verification step is run to verify that the located ellipse is truly an ellipse. This involves computing the average squared distance from the center to the observed boundary points in the image for all θ and comparing it to the ideal value returned given by the parameters found by the algorithm. This version of the HT is iterative when locating multiple ellipses in an image.

2.1.1.4 Fast Ellipse Hough Transform

[26] presents an adaptation of the Fast Hough Transform (FHT), called the Fast Ellipse Hough Transform (FEHT), based on the chord-tangent method shown in Figure 2.2. A set of lines is generated from which the center of the ellipse can be located using the recursive division of the parameter space into hypercubes from lower to higher resolution, according to the process of the FHT outlined in [52]. Next, by using the estimated center, the orientation and the ratio of major to minor axes is computed using another voting method which only includes points

used in the determination of the center in the previous stage. Finally, again using the estimated center, the orientation, and the major to minor axis ratio, the final parameters, a and b, can be determined. Although this algorithm can be parallelized because each hypercube can be processed independently in the first stage, it is recursive in nature, and each stage relies on the output of the previous stage. Additionally, the memory and computation requirements are dependent on the number of edge points of the ellipse (and ultimately on the number of edge points in the image).

2.1.1.5 Elliptical Hough Transform

The Elliptical Hough Transform (EHT) [28] is a hierarchical approach to ellipse detection using the SHT. First, an image pyramid is created, halving the resolution of each dimension of each successive layer of an edge (binary) input image, until the smallest image is created (32×32 in this case). Then, starting from the smallest image, the SHT for ellipses is applied on the image, and votes are accumulated in the 5-D parameter space containing the (x, y) coordinates of the foci and the constant sum of the distance from any point on the ellipse to the foci. After vote accumulation, the accumulator is normalized in order to account for the pyramidal down-sampling process and then candidate ellipses are stored and sorted based on normalized votes. The last step at each image size is to remove duplicate votes. When the image size is increased to the next level, the parameters from the lower stage are used to estimate the location of the ellipse in the larger image and reapply the SHT in that region. A multipass version of the algorithm is also proposed, where detected ellipses are removed from the edge image. The computational complexity claimed by the authors is $\Theta(n^{\frac{5}{2}})$, where n is the dimension of the side of a square input image, and a power of 2. The accuracy of the multipass version of the algorithm is between $80 - 90\%$ when six ellipses are in an image, diminishing when more ellipses are present.

2.1.2 RANSAC

Random Sample Consensus (RANSAC) [53] is a model fitting paradigm designed to be robust in the presence of gross errors, which are errors that do not fit with the assumed noise

model. Typical model fitting methods assume the noise in the data is due to measurement error, which can be accounted for by a noise model, and thus will try to use as many points as possible when fitting a model to the data. However, feature identification in image analysis also has classification errors, which cannot be modeled by standard noise models. This means that these kinds of model fitting methods fail in the presence of gross errors. RANSAC works by randomly selecting the smallest number of data points required to compute model parameters, computes the model, and counts the number of the data points that are within some error tolerance of the model. This process is repeated until a consensus set containing enough points to exceed some threshold (a parameter to the algorithm) is found, or until this process has occurred a certain number of times.

[29] uses standard RANSAC to select five points from a set of points forming an ellipse (predetermined by a different computation method). The points are used to solve a linear system of equations to find the ellipse parameters, which are then used as initial estimates into an iterative least squares approach to improve the estimates.

[31] demonstrates an algorithm using RANSAC to fit ellipses to arc segment groups. First, line segments are extracted from edge pixel information in the image. Then, line segments that potentially are elliptic arcs are linked to make arc segments. Next, arc segments from the same ellipse are grouped together, and, finally, a variation of RANSAC, where inlier segments are sought from both the arc group set as well as the whole image, is used to fit ellipses to these groups.

[54] introduces an algorithm for ellipse detection which extracts edge contours from an image and applies RANSAC to the inflection points of the contours to obtain estimates of the ellipse parameters for each contour. The parameters are then evaluated to check if they agree with image data. The method uses the standard RANSAC approach for ellipses, that is, five points from a contour are selected and the conic parameters for these points are computed. If the conic represented is not an ellipse, then that model fails. In order to avoid problem cases which cause this particular application to fail, such as overlapping ellipses and short arcs and line segments, the authors introduce a fitting factor into the RANSAC algorithm based on the number of feature points which lie on a circle.

[30] proposes a method of ellipse detection based on [54] but uses three points rather than five. First, contours are located, then the edges are thinned to a width of one pixel to reduce the number of extra features. Then, RANSAC is used to select three pixels to fit the ellipse. The three-point method uses the chord-tangent method (Figure 2.2) to locate the center of the ellipse, but modifies the tangent computation. Here, a line is fit by the least squares method for a 5×5 window centered at each edge point. Once the center is located, the major and minor axes and orientation can be computed. In order to obtain deterministic results from RANSAC, the authors propose seeding a pseudorandom number generator with the number of points in the contour so that RANSAC will return consistent results for the same input image.

[55] applies RANSAC to edge segments in an image in order to find elliptic arcs, rather than whole ellipses. They use standard RANSAC, selecting five points from a segment and attempt to fit an elliptic arc to it, and once a segment is fitted, it is removed from the input image. After applying RANSAC to the feature points, they compute a similarity metric for all the detected ellipses and fit a single ellipse to arcs that have high similarity.

2.1.3 Fuzzy Clustering

Fuzzy clustering is an approach to clustering data points such that some points may belong to multiple clusters. [32] proposes two alternatives to the standard fuzzy c-means (FCM) algorithm, due to the inadequacy of FCM when applied to locating curved shapes. Fuzzy c-shells (FCS) identifies circles, and Adaptive FCS (AFCS) identifies ellipses. Both FCS and AFCS contain an additional term in the objective function that represents the ratio of the distance of some point x_k from the cluster center to the distance of x_k from the cluster shell. This ratio is intended to measure the scatter with respect to the shell cluster prototype.

[34] modifies the AFCS variant mentioned above by using the radial distance rather than the normalized radial distance used in AFCS. This is to compensate for a perceived weakness in the AFCS method where fitted ellipses tend to favor "rounder" ellipses, as opposed to oblong ellipses. This improvement increases the computational burden of the algorithm over AFCS. This algorithm is labeled as fuzzy c-ellipsoidal shells (FCES). The authors claim both AFCS and FCES methods require use of the Levenberg-Marquardt (LM) (or similar) algorithm to

update the parameters each iteration of the algorithm. This algorithm can be computationally intensive, but good initialization values help reduce the number of iterations.

[33] also incorporates ellipse parameters into the objective function, using the major axis and the foci in the typical squared error function. The fuzzy k-ellipses (FKE) algorithm typically converges to a local minimum, so to push the algorithm towards the global minimum, three stages are used: first, the fuzzy k-means (FKS) algorithm is used to converge the prototypes to centers, then another variant of FKS called fuzzy k-rings (FKR) is used to converge clusters to circles, finally, the proposed objective function is used to converge the clusters to ellipses.

2.1.4 Kalman Filter

The Kalman filter is useful for smoothing noisy data and estimating unknown parameters. It is a two step process, where in the *prediction* step, estimates for the state variables and their uncertainties are generated, and in the *update* step, the observed measurements are incorporated into the estimates. Typically, the standard Kalman filter uses linear functions for the state transition and observation models, whereas the Extend Kalman filter (EKF) can use nonlinear functions after linearizing the equations. [35] makes use of the EKF to fit ellipses by linearizing the maximum likelihood formulation instead of using the measurement equation.

2.2 Optimization Methods

Optimization problems seek to find the minimum or maximum of some objective function subject to certain constraints. Several methods for ellipse detection fall into this category, such as least squares methods, maximum likelihood methods, and genetic algorithms. This section describes some of these approaches.

2.2.1 Least Squares

Least squares (LS) methods seek to minimize some error function relating the observed noisy data points to the estimated points of the "true" ellipse that would be observed in the absence of noise. This form of ellipse detection then becomes a statistical problem involving estimating

the true points of some ellipse for the given data [56]. There are two main approaches to LS ellipse fitting: algebraic and geometric.

Both methods rely on the general equation for a conic given in Equation 2.1. However, [42], [56] rewrite Equation 2.1 as follows, in order to account for computing a solution using numerical methods when x and y are greater than 100 on machines with finite precision:

$$Ax^2 + 2Bxy + Cy^2 + 2f_0(Dx + Ey) + f_0^2 F = 0 \tag{2.3}$$

where f_0 is of similar order as x and y. We will use the notation of [42], [56] in the following discussion, as it provides a convenient way to discuss differences in the LS methods in spite of the notational disparities throughout the literature.

The basic LS minimization of the algebraic distance of the set of points (x_α, y_α), $\alpha = 1, \cdots, N$, is according to the following equation:

$$J = \frac{1}{N} \sum_{\alpha=1}^{N} \left(Ax_\alpha^2 + 2Bx_\alpha y_\alpha + Cy_\alpha^2 + 2f_0(Dx_\alpha + Ey_\alpha) + f_0^2 F \right)^2 \tag{2.4}$$

The trivial solution $J = 0$ when $A = B = \cdots = F$ can be avoided by applying some form of normalization to Equation 2.3. Various normalization schemes have been proposed in the literature and are shown below:

$$F = 0 \tag{2.5}$$

$$A + C = 1 \tag{2.6}$$

$$A^2 + B^2 + C^2 + D^2 + E^2 + F^2 = 1 \tag{2.7}$$

$$A^2 + B^2 + C^2 + D^2 + E^2 = 1 \tag{2.8}$$

$$A^2 + 2B^2 + C^2 = 1 \tag{2.9}$$

$$AC - B^2 = 1 \tag{2.10}$$

Let

$$\theta = \begin{pmatrix} A & B & C & D & E & F \end{pmatrix}^T \tag{2.11}$$

and for some normalization matrix $\mathbf{N}$, it can be seen that Equations 2.5-2.10 can be written as

$$\theta(\mathbf{N}\theta)^T = 1 \tag{2.12}$$

For the normal LS method, $\mathbf{N} = \mathbf{I}$.

Now, if we let

$$\xi = \begin{pmatrix} x^2 & 2xy & y^2 & 2f_0x & 2f_0y & f_0^2 \end{pmatrix}^T \tag{2.13}$$

We can write the general algebraic LS method as follows and solve for the unit eigenvector θ using the smallest eigenvalue λ:

$$\mathbf{M} = \frac{1}{N}\sum_{\alpha=1}^{N} \xi_\alpha \xi_\alpha^T \tag{2.14}$$

$$\mathbf{M}\theta = \lambda\theta \tag{2.15}$$

where $\mathbf{M}$ is a 6×6 matrix.

In order to account for noise in the model, we can define

$$x_\alpha = \bar{x}_\alpha + \Delta x_\alpha,\ y_\alpha = \bar{y}_\alpha + \Delta y_\alpha \tag{2.16}$$

where $\bar{x}_\alpha$ and $\bar{y}_\alpha$ are the true values of the ellipse, x_α and y_α are the observed data points, and Δx_α and Δy_α are the deviations from the true points due to noise. Assuming a Gaussian distribution for Δx_α and Δy_α with $\mu = 0$ and standard deviation σ, Then the variance is

$$\mathbf{V}[\xi_\alpha] = \sigma^2 \mathbf{V}_0[\xi_\alpha] \tag{2.17}$$

and the normalized covariance matrix $\mathbf{V}_0[\xi_\alpha]$ can be defined as (see [56] for details)

$$\mathbf{V}_0[\xi_\alpha] = 4\begin{pmatrix} \bar{x}_\alpha^2 & \bar{x}_\alpha\bar{y}_\alpha & 0 & f_0\bar{x}_\alpha & 0 & 0 \\ \bar{x}_\alpha\bar{y}_\alpha & \bar{x}_\alpha^2 + \bar{y}_\alpha^2 & \bar{x}_\alpha\bar{y}_\alpha & f_0\bar{y}_\alpha & f_0\bar{x}_\alpha & 0 \\ 0 & \bar{x}_\alpha\bar{y}_\alpha & \bar{y}_\alpha^2 & 0 & f_0\bar{y}_\alpha & 0 \\ f_0\bar{x}_\alpha & f_0\bar{y}_\alpha & 0 & f_0^2 & 0 & 0 \\ 0 & f_0\bar{x}_\alpha & f_0\bar{y}_\alpha & 0 & f_0^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.18}$$

2.2.1.1 Bookstein

[57] uses the normalization method in Equation 2.9 to fit general conics by solving a rank-deficient generalized eigensystem using block decomposition [58].

2.2.1.2 Taubin

[59] seeks to fit implicit curves and surfaces to data for object recognition, object position estimation, and object segmentation using an approximate mean squared distance function which can be solved as a generalized eigenvector problem. [56] interprets this method in the context of the framework given above as

$$\mathbf{N} = \frac{1}{N}\sum_{\alpha=1}^{N}\mathbf{V}_0[\xi_\alpha] \tag{2.19}$$

2.2.1.3 Direct Least Squares

[36] introduces the constraint in Equation 2.10 to obtain ellipse-specific fits. $\mathbf{N}$ ($\mathbf{C}$ in the original paper) becomes

$$\mathbf{N} = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.20}$$

and $\mathbf{M}$ (called the *scatter matrix* in the original paper) becomes an $N \times 6$ matrix (N is the number of points to fit) satisfying

$$\mathbf{M} = \mathbf{D}^T\mathbf{D} \tag{2.21}$$

where the design matrix $\mathbf{D}$ is defined as

$$\mathbf{D} = \begin{pmatrix} \xi_1 & \xi_2 & \cdots & \xi_N \end{pmatrix}^T \tag{2.22}$$

[40] points out that this method for fitting ellipses can be numerically unstable and can produce incorrect results when computing the eigenvalues. In particular, $\mathbf{N}$ is singular and $\mathbf{M}$ is nearly singular (ill-conditioned). As a result, the optimal eigenvalue returned by the method of [36] could be a small negative number. Additionally, when all data points lie exactly on an ellipse (no noise), the eigenvalue is zero. In these cases, the original DLS method will produce non-optimal or incorrect solutions. In order to overcome these stability problems, [40] proposes

the following modifications. First, the $\mathbf{D}$ matrix is split into quadratic and linear parts, such that $\mathbf{D} = (\mathbf{D_1}|\mathbf{D_2})$. $\mathbf{M}$ can be split so that

$$\mathbf{M} = \left(\begin{array}{c|c} \mathbf{M}_1 & \mathbf{M}_2 \\ \hline \mathbf{M}_2^T & \mathbf{M}_3 \end{array}\right) \tag{2.23}$$

Likewise, $\mathbf{N}$ can be split:

$$\mathbf{N} = \left(\begin{array}{c|c} \mathbf{N}_1 & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array}\right) \tag{2.24}$$

Lastly,

$$\theta = \left(\begin{array}{c} \theta_1 \\ \hline \theta_2 \end{array}\right) \tag{2.25}$$

The system can then be expressed with the following equations:

$$\mathbf{N}_1^{-1}(\mathbf{M}_1 - \mathbf{M}_2\mathbf{M}_3^{-1}\mathbf{M}_2^T)\theta_1 = \lambda\theta_1 \tag{2.26}$$

$$\theta_1^T\mathbf{N}_1\theta_1 = 1 \tag{2.27}$$

$$\theta_2 = -\mathbf{M}_3^{-1}\mathbf{M}_2^T\theta_1 \tag{2.28}$$

2.2.1.4 Enhanced DLS

[41] builds on the work of [40] to improve the weaknesses in the DLS method, particularly its inability to determine a solution when there is low noise in the data. It is observed that the construction of $\mathbf{M}$ in DLS using modern image dimensions results in an intrinsically ill-conditioned matrix. In order to overcome this problem, scaling terms are introduced and applied to the data points as an affine transform such that the transformed points will be centered about the origin and contained within a square of side length 2. Once the eigenvector problem is solved, the coefficients must be denormalized. Additionally, if localizing the solution to the eigensystem fails, it is suggested to add known Gaussian noise to the data and recompute the result. Establishing the number of replicates beforehand can avoid iterations when adding noise.

2.2.1.5 HyperLS

[42] derives the HyperLS method from statistical analysis of the LS problem for fitting ellipses. The value of $\mathbf{N}$ is derived in order to compute the coefficients $A, \cdots, F$ as closely to the true values as possible, regardless of the type of conic. This theoretical analysis produces the following value of $\mathbf{N}$:

$$\mathbf{N} = -\frac{1}{N^2}\sum_{\alpha=1}^{N}\left(\xi_\alpha^T \mathbf{M}_5^- \xi_\alpha \mathbf{V}_0[\xi_\alpha] + 2\mathcal{S}[\mathbf{V}_0[\xi_\alpha]\mathbf{M}_5^- \xi_\alpha \xi_\alpha^T]\right) \tag{2.29}$$

where $\mathbf{M}_5^-$ is the pseudoinverse of $\mathbf{M}$ of truncated rank 5 computed by the eigenvalues of $\mathbf{M}$, denoted $\mu_1, \cdots, \mu_6$ and satisfying $\mu_1 \geq \cdots \geq \mu_6$. The corresponding unit eigenvectors are denoted $\theta_1, \cdots, \theta_6$. $\mathbf{M}_5^-$ can then be written as

$$\mathbf{M}_5^- = \frac{1}{\mu_1}\theta_1\theta_1^T + \cdots + \frac{1}{\mu_5}\theta_5\theta_5^T \tag{2.30}$$

$\mathcal{S}[\cdot]$ is the symmetrization operator, defined for some matrix $\mathbf{A}$ as

$$\mathcal{S}[\mathbf{A}] = \frac{\left(\mathbf{A} + \mathbf{A}^T\right)}{2} \tag{2.31}$$

[42] also finds that all the LS estimators have similar performance when the data points cover more than 25% of the circumference of the ellipse, but for short sequences of points, the standard LS approach becomes unreliable. Additionally, the differences between the outputs from the Taubin method and HyperLS method are minute, but HyperLS tends to fit better ellipses.

2.2.1.6 Geometric Methods

Geometric LS methods seek to minimize the sum of squares of the orthogonal distance from the observed points to the ellipse. Again, using the notation from [56], this can be written as

$$S = \frac{1}{N}\sum_{\alpha=1}^{N}\left((x_\alpha - \bar{x}_\alpha)^2 + (y_\alpha - \bar{y}_\alpha^2)\right) \tag{2.32}$$

In this equation, $\bar{x}$ and $\bar{y}$ are variables estimating the true values of x_α and y_α. Note that this is inherently a nonlinear problem and these types of problems typically require iterative methods to compute a solution.

If the distance from the points to the ellipse is small, the geometric distance can be approximated by the Sampson Error, and Equation 2.32 becomes

$$J = \frac{1}{N}\sum_{\alpha}^{N}\frac{(\xi_{\alpha}^{T}\theta)^2}{\theta^T\mathbf{V}_0[\xi_\alpha]\theta} \tag{2.33}$$

There are two iterative methods to minimize this equation: the Heteroscedastic Errors-in-Variables (HEIV) method and the Fundamental Numerical Scheme (FNS) method.

Other variants of geometric methods formulate the problem in different ways, but ultimately are seeking to minimize the orthogonal distance from the observed points to the ellipse. For example, [39] uses a parametric form of the ellipse equation and solves $2N$ equations with $N+5$ unknowns using N points. In order to solve this nonlinear system they utilize several variants of common iterative solvers: the Gauss-Newton method, the Newton method, the Gauss-Newton with Marquardt modification method, the variable projection method, and the orthogonal distance regression method (ODR). In their problem formulation, the ODR performs the best.

[43] seeks to compensate for some of the computational inefficiencies of [39] by using the Cartesian representation of an ellipse and the orthogonal contacting conditions between a measured point and the fitted ellipse. Additionally, similar to [39], the initial parameter vector for the Gauss-Newton method is from first fitting a circle to the data points, but without the singularity introduced by the formulation in [39].

[60] proposes a method for geometric fitting combining two operators, one linear and the other nonlinear. The basis of the method uses the geometric model of the ellipse and formulates the LS component using the distance from pixels to the fitted ellipse. This ElliFit method seeks to solve a set of equations in order to obtain the geometric parameters of the ellipse rather than for the general conic parameters, that is, ElliFit solves for the major and minor axes lengths, the orientation, and the center coordinates. The method requires at least $N = 7$ points to ensure success, and the authors claim the computational complexity of their method is $O(N)$, where N is the number of input points.

2.2.1.7 Maximum Likelihood

Maximum likelihood (ML) formulations are similar to the geometric distance formulations. According to [56], when the noise distribution of the observed points is assumed to be Gaussian, the ML methods and geometric distance methods are equivalent. Indeed, the process of [46] for fitting ellipses based on the ML formulation looks similar to the geometric methods above, utilizing the FNS method to extract the ellipse parameters.

2.2.2 Parallel Chord Method

[44] proposes an alternative to algebraic LS methods that uses parallel chords to fit ellipses to points. By using the chord-tangent method to locate the ellipse center, the other parameters of the ellipse can then be computed. The procedure starts by collecting a set of three parallel chords perpendicular to some axis (e.g., the y-axis) and plotting the quadratic polynomial representing the chord lengths as a function of their positions from this axis. Then, a 3×3 linear system of equations is used to compute the coefficients of the best quadratic fit for the chord data. Next, using a plot of the positions of the midpoints of the chords as a function of y, a 2×2 linear system of equations is solved to obtain the coefficients of the line of best fit. Using the coefficients from these solutions, the parameters can be computed directly. To guarantee an elliptic fit, it is sufficient to add a constraint to the quadratic fitting in order to ensure the parabola opens down, which can be solved by using a 1-D quadratic programming minimization problem. [44] claims this method performs better than DLS in terms of the bias of the fitted ellipse and in the case of occlusion, and can do so with fewer computations.

2.2.3 Genetic Algorithms

Genetic algorithms (GA) are a type of optimization algorithm used to explore the solution space of complex problems by emulating biological processes. A set of potential solutions, each with a set of properties (*chromosomes*) form a population that is modified each iteration of the algorithm by selection of a subset of the "fitter" solutions, and between pairs of these solutions, a *crossover* operation is applied to mix properties of both solutions together to produce two

offspring, under the assumption that fitter *parent* solutions will be more likely to produce solutions nearer the optimal solution. Solutions which do not exceed the fitness threshold are culled. A *mutation* operation is applied randomly to solutions to help prevent stagnation of the solution space. A *fitness function* is used to measure the quality of a solution that solves the problem.

[45] proposes a two-stage ellipse detection algorithm where the first phase is a GA using the ellipse parameters to produce a list of candidates, and the second phase improves the fitness values of selected solutions. The solutions are binary strings representing the ellipse center coordinates, major and minor axes lengths, and orientation. The fitness function is defined as

$$f(S) = \sum_{(x,y),\in L_s} e(x,y) \tag{2.34}$$

where $e(x,y)$ is the value of a pixel at the given location in the edge image for the set of points, L_s, defined by the parameters in the solution string S. At first glance, this could be a significant number of points, but the authors exploit shape symmetry to reduce the number of points to four. In order to select the solutions which will "reproduce," the method first normalizes the fitness values of the current generation by dividing $f(S)$ by the average fitness of the current generation. A modified roulette-wheel selection process then takes place. An arithmetic crossover scheme which linearly combines the parameters of the parents is used when both fitness values of the selected solutions are greater than the generational average fitness value. If the fitness values of both parents are not greater than the generational average, then offspring are produced by interchanging substrings at a random location. This method also employs a mutation mechanism. After recomputing the fitness function, the solutions with values above a certain threshold are added to a list, as long as the difference between current solutions on the list is greater than a distance metric using the fitness value. This process is repeated for a certain number of generations. Once the GA stage is completed, each solution on the list is re-evaluated for fitness by perturbing each parameter value by some amount and exchanging the current value with the value from every other solution in the list.

[7] uses a set of five points as a chromosome. In order to avoid redundancy in the population, identical chromosomes are removed, that is, if two chromosomes have the same geometric

parameters, they are regarded as equal. The fitness measure of the algorithm is based on the similarity of candidate ellipses, which measures how close the perimeter of a candidate matches the perimeter of an ideal ellipse, and the distance metric, which characterizes the distance between the candidate ellipse perimeter and the ideal ellipse perimeter. A desirable candidate has "good" similarity and small distance, or has "acceptable" similarity and excellent distance. This method also utilizes a multi-population approach, which means the initial population may split into several subpopulations which progress independently. A subpopulation may terminate in the case of one of three conditions: an optimal chromosome exists, as defined by thresholds for the similarity and distance values, or if a good chromosome is the best in a subpopulation without improvement for 30 generations, or 500 generations elapse without the fulfillment of the first two conditions. The algorithm also includes three population operations based on a Euclidean distance measure to improve the progression towards local and global optima. They are migration, where a chromosome can move to another subpopulation, splitting, where a migrating chromosome splits from a subpopulation but cannot locate another close subpopulation (according to the distance metric) so it starts a new subpopulation, and merging, where two close subpopulations come together into a new subpopulation. This method utilizes elitism and fitness-proportional selection mechanisms, and crossover and mutation mechanisms for diversification. The crossover mechanism is a simple random substring selection and the parents' points are swapped to form two new chromosomes. When mutating a chromosome, a point from the chromosome is randomly selected, which then becomes the starting point of a path traversing the perimeter of a pattern until a set number of points are added, or the end of the pattern or an intersection of patterns is encountered. Four points from this set are then randomly selected and added to the initial point to form a new chromosome.

2.3 Summary

This section describes the basic algebraic and geometric features of ellipses which are exploited by computer vision algorithms in order to determine if line segments and contours in images are from ellipses. As can be observed, there are many conceivable ways to locate ellipses in images, and this chapter outlines only several of the major methods proposed in the

literature over the years. Many of these methods are iterative, or computationally or memory intensive and are therefore not necessarily suited to run unmodified in a real-time embedded system. The next chapter outlines some of the related work where certain of these algorithms have been accelerated in some way in order to improve run-time performance.

CHAPTER 3. RELATED WORK

This chapter discusses implementations seeking to improve the performance of the ellipse detection algorithms outlined in the previous chapter. Examples of implementations on GPGPUs, FPGAs, DSPs, embedded CPUs, and SoCs are highlighted, but those on workstations are excluded, since they often function as the baseline comparison.

3.1 General Purpose Graphics Processing Units (GPGPU)

One technique to take advantage of parallelism present in an algorithm is to use a SIMD architecture, such as a GPGPU. While not yet popular in embedded real-time systems, recent advances in virtual and augmented reality are demonstrating the applicability of embedded GPGPU architectures. Therefore, in order to provide some reference for possible future mobile GPU implementations, several GPGPU ellipse detection algorithms are described below.

[61] uses a GPGPU for fitting ellipses with the HT. This method decomposes the 5-D parameter space necessary for detecting ellipses using the SHT into three 2-D parameter spaces, one for determining the center coordinates, one for determining a slope, and the last for determining the semiaxes. After voting in the final parameter space, the detected ellipses are validated using a Euclidean distance map approach, where each location of the map stores the distance to the closest edge pixel. Since this algorithm is only able to locate one ellipse in an image at a time, the image is split into overlapping sub-images. The algorithm processes 2048×2048 pixel images using an Nvidia GeForce GTX 480 GPGPU. The work uses previously implemented kernels for Canny edge detection and Euclidean distance map generation, and then implements a kernel to generate an edge list from an edge image. The voting process is accomplished through another kernel, and an evaluation kernel checks the ellipse candidates

against the Euclidean distance map. Although the authors claim a 64.79× speed-up over an equivalent (sequential) CPU version of the algorithm, the performance of the GPGPU version is only 1.3 FPS, which demonstrates this is not an efficient implementation of the HT, even though a high-throughput architecture is used.

[62] uses a GPGPU for fitting ellipses with a variant of the RHT utilizing a linear least squares step to fit a general conic equation to the set of five points. If the conic is an ellipse, the center, semiaxes lengths, and orientation are computed and voted for in three parameter spaces, two of which (center and lengths) are 2-D, and the last is 1-D. The set of boundary pixels is given as input to the GPGPU algorithm, as well as several random numbers. Randomization of the input points is achieved by shuffling the texture memory where the boundary points are stored using the random numbers given to the GPGPU. The algorithm generates 10,000 conic equations and computes the centers, semiaxes lengths, and orientation for each. These results are returned to the CPU, where voting and peak-finding occur to determine the final ellipse parameters. The authors claim the communication overhead back to the CPU software is negligible. This algorithm was tested on an NVIDIA GeForce 7950GT GPGPU using a range of image sizes up to 1200×1368 pixels. The performance achieved at this size is reported at 52 FPS, with a speed-up of 9.2 over a CPU-only implementation. This method only detects single ellipses in images. In spite of reasonable speed-up and performance numbers, this implementation still leaves much to be desired. Better performance could potentially be achieved through accelerating vote accumulating and peak finding, and by reducing the communication overhead between CPU and GPGPU. Additionally, the restriction on detecting multiple ellipses limits the usefulness of this approach.

[63] explores two methods to improve performance of the Fast GHT (FGHT) on an NVIDIA GeForce GTX 280 GPGPU, the first to achieve better load balancing, and the second to achieve better occupancy. The FGHT breaks the GHT computation into three transforms to obtain the rotation, scale, and displacement parameters. After Canny edge detection, the edge points of both the template image and the input image are compacted to remove non-edge points, and sorted into lists by the (quantized) gradient value. These contour points are compared against each other and paired if the difference between their gradients is within some distance.

The pairings are used to vote for shape orientation in the orientation parameter spaces of the template image and the input image. Then the correlation between the two orientation parameter spaces is computed, and the maximum correlation value is then used to determine the object orientation. During this phase, the contour pairs are also sorted into new lists by an index value based on the orientation. These lists are used to compute the scaling factor of the object, as well as the displacement factor. The load balancing method attempts to evenly distribute the computational load across all processing elements, and performs best when searching for pairings in the edge lists. The save-shared-memory method seeks to maximize processing element occupancy by reducing the amount of data loaded into shared memory, and performs best when computing the scale and displacement of the object. These results naturally led to an implementation merging the two strategies, the performance of which varies, but can process 4,000 frames from a video with resolution 352×288 pixels on the order of hundreds of milliseconds (over 14,000 FPS). The performance of this method is highly dependent on the input frame for performance, which is mostly irrelevant at such high frame rates.

3.2 Field Programmable Gate Arrays (FPGAs)

3.2.1 RANSAC Architectures

[64] builds an ellipse detection FPGA architecture for locating circular road signs in images using RANSAC. The processing pipeline has four main stages, and begins with a preprocessing stage. This stage first applies a Gaussian noise reduction filter, then histogram stretching for image contrast enhancement, then a Sobel filter coupled with an edge-thinning method to produce 1-pixel wide edges, before finally outputting the pixels of a binary image. The next stage locates the edge pixels in the binary image and stores them in device BRAM for the RANSAC module to read. A double-buffering scheme is used to ensure one set of data is always ready to advance into the RANSAC module. The RANSAC stage randomly selects three points from the data buffer to generate three candidate ellipses by using the tangent-bisection method outlined in Chapter 2, which are passed to the next stage. In the final stage, the models are verified using the data points in the data buffer by counting the number of points that are

within some boundary distance from the candidate ellipse. Because model verification is time consuming, the clock for this last stage is configured to run 9× faster than the other modules. The architecture was configured to compute 1,200 iterations of RANSAC for synthetic images having dimensions of 640×480 pixels. When testing with real images, however, the design only performs at 15 FPS while using 2,500 iterations of RANSAC, and only achieves a detection rate of about 75%. Implementing RANSAC on FPGAs is a complex problem, and the limitations of the detection performance can be linked to the modifications to the algorithm necessitated by providing a result after a fixed time rather than seeking to reach a fixed confidence value with a solution. Further limiting applicability for the sake of performance, the design only detects ellipses whose major axis orientation is either 0° or 90° from the x-axis. Lastly, the difficulty of RANSAC is also demonstrated by other design complexities, such as fixed point computations and multiple clock domains.

[65] builds an improved FPGA RANSAC architecture for ellipse detection to use in a real-time eye-tracking system. There are three major stages in the system: first, the image is processed to prepare for feature extraction, then the features are extracted using the Starburst algorithm, and finally, the ellipses representing the irises are estimated using RANSAC. Before performing the Starburst method, the image must be converted to RGB from a Bayer arrangement, then from RGB to grayscale, then the corneal reflections are removed by a horizontal bilinear interpolation method before a box blur is applied to the image. Next, the Starburst algorithm is performed and has three major stages: first, the gradients for all pixels are computed, then the distances and angles from a base point in the image to the largest gradient along a ray from that point are computed, and these feature points are added to a feature table, which is then trimmed by removing invalid feature points. After the Starburst method completes, RANSAC is used to compute ellipse hypotheses using five points at a time from the feature table. The RANSAC portion is divided into two stages: the hypothesis generation stage, and the ellipse parameter computation stage. The hypothesis generation stage builds a system of equations using the five randomly selected points, and then one of three solvers is used to solve the system of equations: Cramer's rule, Gauss-Jordan elimination, and Doolittle LU decomposition. It was found the choice of solution was not detrimental for the quality of

the RANSAC solution, nor did it impact execution time. The design operates at a rate of 62.5 FPS, as long as the design clocks (there are three domains) exceed the 25 MHz camera clock. No resolution information is given. Power dissipation of the system was measured using each solver, and the FPGA dissipated power in the range 3.16-3.34 W. This design overcomes some of the implementation difficulties pointed out in the previous work, and demonstrates the feasibility of mixed streaming and iterative architectures, but still requires more than one clock domain in order to function.

3.2.2 Hough Transform Architectures

The SHT has a rich history of acceleration via dedicated hardware and FPGA designs (see [66] for an early survey), but, to the best of our knowledge, very little work has been done to accelerate the HT specifically for ellipse detection using an FPGA. [67] discusses some of the complexities that arise when implementing the SHT on FPGAs, and mentions two main challenges for FPGA implementations of the SHT. It is known voting in accumulator space requires significant bandwidth in order to cast votes (a read and write for each bin), ergo the first difficulty arises from the limited memory bandwidth available on typical FPGAs. There are methods to overcome some of the bandwidth challenges, such as coarsening the accumulator space, potentially requiring multiple passes through the image to obtain the necessary accuracy, which is difficult for a streaming-architecture system. Alternatively, off-chip memory may be used for the accumulator space, but access is significantly slower, which may not be acceptable if the system must operate quickly. The second major challenge is the complexity of the computations themselves. Many floating point multiplications need to be performed, as well as trigonometric functions such as sine and cosine, but these types of computations are not trivial on FPGAs. Alternatives, such as CORDIC architectures [68], logarithmic number systems [69], and incremental sums [70], [71] have been proposed to alleviate some of the computational complexity, but these methods still have drawbacks and may not reduce the difficultly of using the HT for real-time ellipse detection.

Although not ellipse-specific, the GHT can still be used to detect ellipses simply by building appropriate tables for the shape. An FPGA implementation of the GHT is given in [72], which

uses a two-phase design to process images with dimensions 640×480 pixels. The architecture replaces the standard R-tables with Shape Tables and implements double buffering using two external memory banks, allowing one image to be written while another is processed. Edges are first detected in the image, and the addresses of these edge points are stored in another memory bank. These addresses are then read into the first phase accumulator unit, which uses the addresses to read out Reduced Shape Tables from yet another memory bank and adds the values in the tables to the accumulator. Next, the M largest values in the first accumulator are found, which will form sub-regions in the original accumulator array, and are used as likely regions where an object center lies. In the second accumulator unit (the second phase), the edge addresses are read again from memory and used to add the original Shape Tables to the M sub-regions from the first phase. The accumulator array is then searched for peaks, and the peaks correspond to object centers. The performance for this design is heavily dependent on the number of edge pixels located in an image. For images with about 2.8% of edge pixels, the rate is 26.3 FPS, but when 6% of pixels are edge pixels, the rate drops to 13.3 FPS. However, [66] estimate the average number of edge pixels in a typical image to be 10%, making this design unrealistic. The significant dependence on the number of edge pixels, as well as the reliance on external memory banks make this design unsuited for an embedded real-time system.

3.3 Digital Signal Processors (DSPs)

DSP devices are similar to FPGAs, but are more specialized for digital signal processing domains, and typically contain many multiply-accumulate units (MACs). While not as common for implementing computer vision algorithms, [18] uses an ellipse-fitting technique for impedance measurement, which fits an ellipse to two sinusoidal signals captured simultaneously and translated into the xy plane. A fitting technique based on [40], [58] (see Chapter 2) is used with modifications to reduce memory requirements. Deployed on an Analog Devices ADSP-BF533 DSP (Blackfin-based) clocked at 594 MHz, the design acquires 960 samples and applies the ellipse fitting algorithm in 28.3 ms. Although not locating ellipses in images, if the samples were considered as edge pixels, then the equivalent processing rate is about 35.7 FPS, comparable to other real-time embedded systems, thus demonstrating potential benefits

of using DSPs in ellipse detection systems. This work also displays the utility and generality of the ellipse-fitting methods.

3.4 Embedded CPU

[73] develops a Raspberry Pi-based ellipse detection system (rev. B+) in order to monitor bees entering and leaving a hive. In order to facilitate rapid counting, the bees are modeled as ellipses. The system is set up to capture 30 seconds of video every 10 minutes at 1920×1080 resolution and 5 FPS. The videos are processed and then the results sent to a database via the Internet. OpenCV is used for the necessary image processing functions. Because of the limited scope of the problem (stationary camera, known distance, standard bee size), bees were able to be modeled as ellipses of fixed size, and the system only needed to estimate the centroid and orientation of bees using a thresholded input image. A linear regression model was also developed to estimate the number of bees when a cluster was detected. This system segments and counts the ellipses at a rate of about 2.6 FPS, and measures bee in-and-out flow at about 0.53 FPS. Because of the system application requirements, this rate difference was acceptable, provided the entire video sequence was processed within ten minutes. Although this system is an example of applied embedded ellipse detection, it does not quite qualify as a real-time system, due to capturing an entire video before processing the data rather than frame-by-frame during operation. However, it does demonstrate the possibility of, and challenges for, ellipse detection using embedded CPUs.

[74] observes that many of the ellipse detection algorithms are unsuited for embedded devices, such as mobile phones. To overcome this problem, an algorithm is proposed to extract arcs from an image, classify each arc based on convexity, then group arcs into threes based upon the convexities, mutual position, and estimated centers from each arc. After the grouping, a modified HT is performed using three 1-D accumulators, one for the semiaxes ratio ($\frac{b}{a}$ in Figure 2.1), one for the orientation, and the last for the value of the major axis (a in Figure 2.1). Lastly, the fit ellipses are validated against the data, and duplicate detections are removed. The method was tested on a Samsung Galaxy S2 using an Android application which calls the C++ implementation of the algorithm using the Java Native Interface. The accuracy of the

method could not be tested this way, but required capturing the scenes from the phone and testing effectiveness offline. It is not clear from the text which version of the S2 is being used, but the CPUs from both variants are similar, effectively an ARM Cortex A9 clocked between 1.2 and 1.5 GHz. The authors report an average processing time on the S2 of 45.82 ms, which corresponds to a frame rate of about 21.8 FPS on images with resolution of 640×480 pixels. The ability of this algorithm to detect ellipses is good in general, but breaks down when there are small, partially occluded, or very elongated ellipses in the image. Although the run-time performance of this method is reasonable, it is not quite sufficient for our purposes. The FPS value is not fast enough to run in real-time (despite the authors' claims), and cannot locate partially occluded ellipses, or small ellipses, both of which may occur in our environment. While this is the only paper, to the best of our knowledge, that tests an ellipse detection method on a smartphone, it demonstrates that an embedded CPU alone may not be enough to achieve the desired performance.

3.5 System-on-Chip (SoC)

[21] builds a smart-camera system based on the Zynq SoC for use in an intelligent transportation system, in this case, for a camera near a roadway or intersection. Several algorithms are implemented to accomplish four major tasks: vehicle queue length estimation, vehicle detection for counting and speed estimation, vehicle type recognition, and vehicle color recognition. The system can process frames at a rate of 50 FPS for images having dimensions of 720×576 pixels. A pixel-streaming architecture is used for image processing tasks in the FPGA fabric (fine-grained architecture), whereas frame-level processing (coarse-grained image processing architecture) takes place on the ARM CPU. The system runs PetaLinux on the CPU. Several global operations are applied to every incoming image at the front of the pixel stream, including color conversion, Gaussian filtering, Sobel edge detection, a local binary pattern transform module, and consecutive frames differencing. Every frame is then split into three lanes and processed in parallel by duplicate hardware pipelines. Each lane is further split into four regions of interest. The algorithms were prototyped in C++ using OpenCV, then ported to Verilog and simulated, then tested in the reprogrammable fabric. The software application on the

CPU utilizes information obtained from the hardware modules in order to perform high-level image analysis, data transfer, and system control through hardware modules which communicate with the camera and LCD output screen. The design lacks a module for image distortion correction. Although this work does not perform ellipse detection, it is representative of a hardware/software co-design strategy for the Zynq SoC, achieves good results, and a similar methodology is followed in this work.

[75], [76] build an embedded ellipse detection system for micro aerial vehicles (MAVs) to perform relative localization in a swarm of MAVs. The system is built with a Gumstix Overo board containing a Texas Instruments OMAP 3503 processor running at 600 MHz. The OMAP SoC does not have reprogrammable fabric like the Zynq, but instead has an ARM Cortex A8 CPU, a DSP video accelerator, a PowerVR GPU, a display subsystem, a camera interface, and peripheral interfaces such as USB and serial, all on a single chip. Using this SoC, the goal of the MAV system is to identify a target with concentric circles, one black and one white. Ellipse detection is used to locate the circles, because, in the general case, the projection of circular patterns in an image is an ellipse. Detection proceeds as follows: first, pixels in the image are evaluated by color against a dynamic threshold value until a black pixel is located, at which point a flood-fill technique is used to find the bounding box, number of pixels, and centroid of the black region. If the number of pixels and a roundness measure based on the bounding box is approximately what is expected, then the algorithm skips to the computed centroid of the black region and checks if the pixel there is white. If so, a similar process occurs to find the bounding box, number of pixels, and centroid of the white region. If the parameters of the regions are sufficient to infer the target has been found, the center of the ellipses (white and black) are determined by computing the mean of the pixels in each region. The covariance matrix of the pixel positions of each region is computed and used to determine the semiaxes and orientation of each ellipse region by finding the eigenvalues and eigenvectors. Lastly, the circularities of the regions are checked. Because the system uses a heuristic to start the search for the target in a new frame at the location of its last known position, the frame rate of the system varies. The system also is able to operate at different image resolutions, but the maximum reported resolution is 752×480 pixels, and the frame rate ranges from 7-27

FPS. Despite promising results, the maximum image resolution is not large enough for our purposes, and the FPS performance is not able to handle even a 30 FPS camera in real time. Additionally, the algorithm uses a custom ellipse-fitting method that is specific to the problem at hand and may not be robust in the environment or application we are targeting in this work. However, the simple heuristic used is a useful optimization, and a similar process is applied in our algorithm, as will be shown in Chapter 4.

[20] builds a Zynq-based system to detect red or blue road signs. In the FPGA fabric, the pixels are color and gamma corrected through a streaming pipeline architecture to prepare them for further processing. As pixels leave these first several stages, the stream is split such that one branch is written to DRAM without further processing, and the other branch passes through a color filter module in order to replace any color other than red or blue with black, before being written into DRAM. The software then reads the filtered and unmodified images and uses the OpenCV library to apply Canny edge detection to the image, detect contours, fill the contours larger than some threshold with white pixels, apply the circular HT to locate and detect circular signs, and then remove them from the image. The image again passes through contour detection as before to relocate contours after removal of circular shapes. Convex hulls are then detected, and OpenCV uses the Ramer-Douglas-Peucker method to approximate the hull shape by counting corners, then classifying the shape as a triangle, rectangle, octagon, or unknown. Once an approximate shape is determined, it is matched with a sign template using an OpenCV template matching function and the results are output. Although the camera can stream images of dimensions 1920×1080 pixels at 72 FPS, the proposed design operates at a rate in the range of 0.143-0.2 FPS when there are 20 and 0 signs in the image, respectively. In spite of meeting the performance goals given in the paper (maximum of 10 seconds per frame), the frame rate is rather slow, and would not be able to perform well in a real-time embedded situation. However, this work does demonstrate an architecture developed by applying a reasonable hardware/software co-design process, and a similar approach is used in our work to produce a similar architecture, as will be described in Chapter 4.

3.6 Summary

As can be seen from the examples above, GPGPU solutions consume too much power and are not yet suited for embedded systems at this time, although embedded GPGPU algorithms may become viable in the future. FPGA and DSP solutions are useful, but difficult to realize in a full system which mixes a streaming architecture with iterative or random-access architectures. Despite their utility, soft-processor designs on FPGAs are also not fast enough to handle real-time performance requirements. Several embedded CPU-only systems have been attempted, but these too, are not sufficiently fast enough to perform ellipse detection algorithms at the camera frame rate. Finally, several SoC systems have been demonstrated, some with ellipse detection in mind, and some without, but even of these systems, the ones that perform the best are only operating on images with sizes less than what typical cameras currently support. The system described in [20], perhaps the closest to our work, processes full HD images, but at a very slow frame rate. Therefore, an embedded system which is able to detect ellipses in real time is an important contribution, and an embedded SoC may be the best platform architecture that meets many size, weight, and power requirements while still providing adequate computational resources to solve the problem.

CHAPTER 4. THE ALGORITHM

The purpose of the algorithm discussed in the rest of this paper is to determine in each frame the location of the centers of six black circles, arranged in a grid-like pattern on a white background (Figure 4.1). Once the centers have been identified, their precise location in the real world can be recovered. While finding circles in a frame is a well-studied research topic in computer vision, the challenges for this particular algorithm arise from the difficult environmental conditions that exist where the system will be deployed, including severe dust and vibration. The algorithm described in this paper has four major phases, each of which has a number of specific sub-steps. A flow chart is given in Figure 4.2. The rest of the chapter describes the algorithm in more detail.

4.1 Camera Model

Initial development of the system occurred without an actual camera streaming live frames, simply due to lack of resources. To make up for this, it was necessary to emulate a camera

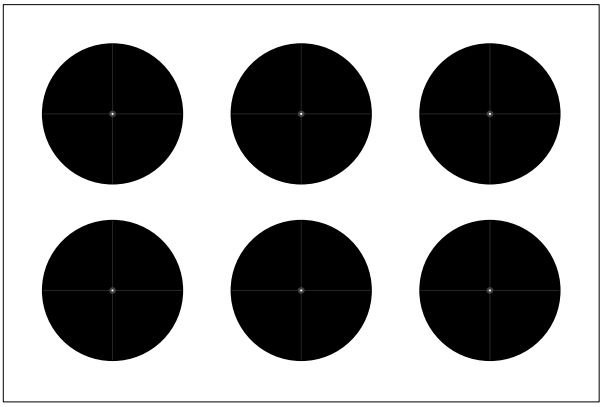


Figure 4.1: The target and markers.

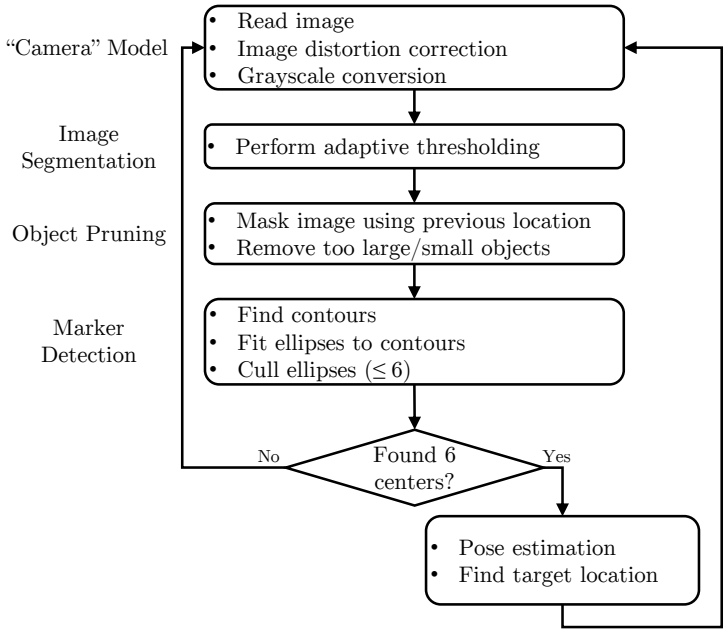
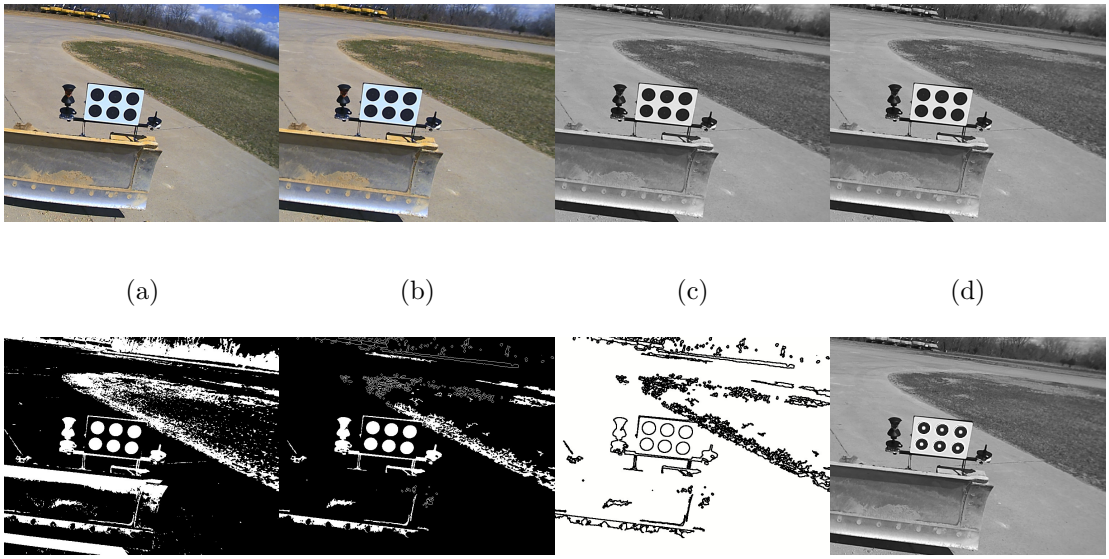


Figure 4.2: The basic algorithm.

pixel stream by loading previously captured raw color images and then treating these images as a sequence of frames. The first steps of the algorithm prepare each frame for processing in later stages.

4.1.1 Distortion Correction

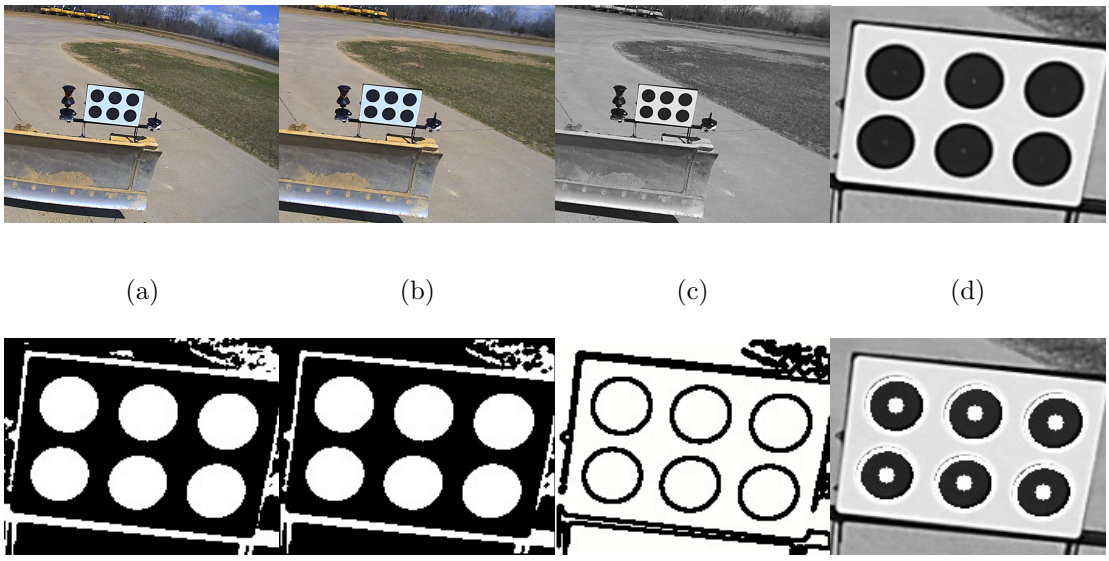
Image distortion arises from the deviation from the ideal pinhole camera model that occurs in real cameras as a result of collecting as much light as possible in a short amount of time through a lens, as well as imperfections in the lens shape, and image sensor construction. There are two major types of distortion: radial and tangential. Radial distortion occurs when a lens bends light entering near the edge of the lens more than light entering near the center, and can be further classified into "fisheye" (or "barrel") distortion, or "pincushion" distortion. The former is where the light is bent such that the edges of the image appear to have lesser magnification than near the optical center. Vertical lines appear to be bowed out near the center in images with this type of distortion. The latter bends light so that the edges of the image appear to have greater magnification than on the optical axis. Vertical lines appear to be bowed inward near the optical center. Tangential distortion results when the lens and



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 4.3: Algorithm applied to a full image. (a) Distorted input image. (b) Image after distortion correction. (c) Image after grayscale conversion. (d) Image after cropping (no effect, in this case). (e) Image after thresholding. (f) Image after blob culling. (g) Image after contour detection. (h) Image with detected centers.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 4.4: Algorithm applied to a cropped image. (a) Second input image (distorted). (b) Image after distortion correction. (c) Image after grayscale conversion. (d) Cropped image. (e) Cropped thresholded image. (f) Cropped image after blob culling. (g) Cropped image after contour detection. (h) Centers in cropped image.

image plane are not parallel. To see an example of distorted images, observe Figure 4.3a and Figure 4.4a, and compare these images with their distortion-corrected counterparts in Figure 4.3b and Figure 4.4b, respectively, and observe the change in the amount of sky visible in the upper right corner of the images.

In order to correct the image distortion, pixels can be remapped by computing new coordinates using functions combining models for radial distortion [77] and tangential distortion [78]. If M_x is the set of remapped x-coordinates, M_y is the set of remapped y-coordinates, and (u, v) is the corrected coordinate, then

$$M_x = \frac{u - c_x}{f_x}\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) + 2p_1\left(\frac{u - c_x}{f_x}\right)\left(\frac{v - c_y}{f_y}\right) + p_2\left(r^2 + 2\left(\frac{u - c_x}{f_x}\right)^2\right) \tag{4.1}$$

$$M_y = \frac{v - c_y}{f_y}\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) + p_1\left(r^2 + 2\left(\frac{v - c_y}{f_y}\right)^2\right) + 2p_2\left(\frac{u - c_x}{f_x}\right)\left(\frac{v - c_y}{f_y}\right) \tag{4.2}$$

where (c_x, c_y) is the optical center (possibly different from the image center), (f_x, f_y) is the focal length of the lens, in pixels, $r^2 = \left(\frac{u-c_x}{f_x}\right)^2 + \left(\frac{v-c_y}{f_y}\right)^2$, the radial distance from the optical center, k_1, k_2, k_3, are the radial distortion camera coefficients, and p_1, p_2 are the tangential distortion camera coefficients. (c_x, c_y), (f_x, f_y), k_1, k_2, k_3, p_1, and p_2 are all computed during a process known as *camera calibration*. Once these maps are computed, then image distortion correction becomes a look-up followed by some interpolation computation:

$$I_{dest}(x, y) = I_{dist}(M_x(x, y), M_y(x, y)) \tag{4.3}$$

where I_{dest} is the corrected image, I_{dist} is the distorted image, and (x, y) is the image coordinate to correct. While this is not the only way to correct for distortion, it is a common approach, and the popular OpenCV library uses this method [79].

4.1.2 Grayscale Conversion

In this stage, the input color image is converted to a grayscale image by doing a simple fixed-coefficient multiplication. In a system with a real camera, this stage still exists, but the

conversion will depend on the output format from the camera device. In our algorithm, the equation is

$$Y = 0.299R + 0.587G + 0.114B \tag{4.4}$$

where R, G, and B correspond to the red, green, and blue channels, respectively, of the input image.

4.2 Image Segmentation

After completing basic image processing steps necessary for preparing the image for further analysis, only one more basic step remains to apply to the whole image, which is to divide the image into two regions: foreground, or areas of important information, and background, which is everything else. Thresholding algorithms are one class of algorithms used to perform this computation.

4.2.1 Adaptive Thresholding

Of the possible thresholding algorithms, there are two types: local and global. Local thresholding algorithms only take into account the pixel values in a neighborhood around the center pixel, whereas global algorithms attempt to use all the information in the image in order to determine if the center pixel is in the foreground or background. Two local methods are explored in this work: Sauvola's thresholding [80] and mean thresholding. Sauvola's method computes the mean and standard deviation for every $w \times w$ window and then determines the threshold value for the pixel located at (x, y) based on the following equation:

$$T(x, y) = m(x, y) \left[1 + k \left(\frac{s(x, y)}{R} - 1 \right) \right] \tag{4.5}$$

where $m(x, y)$ is the mean of the window, $s(x, y)$ is the standard deviation of the window, k is a parameter in the range $[0.2, 0.5]$, and R is either the maximum possible value of the standard deviation (e.g., 128 for an 8-bit grayscale image), or the maximum standard deviation value computed globally, which requires multiple passes through the image. If $I(x, y) > T(x, y)$, where $I(x, y)$ is the value in the input image at coordinate (x, y), the output is "1," otherwise, the output is "0." It is readily apparent that this method in the given form will involve

numerous computations, many of which are repeated. To improve the execution time for the algorithm, [81] uses integral images (summation-tables) to convert the summation operation into two additions and two subtractions.

The second method explored in this work is a local approach based on the mean value of the pixel values in a given neighborhood. If the value of a given center pixel is greater than the mean value of the neighborhood minus some constant offset, the pixel is considered part of the foreground. This method is "adaptive" because the neighborhood size is allowed to change between frames based on the results of the later stages of the algorithm, as well as the value of the constant offset.

4.3 Object Pruning

In order to reduce the number of spurious ellipses detected, as well as improve the processing time of later stages in the algorithm, some amount of processing time is invested to remove extra information from the image. Two methods are used, the first is to mask or crop the image, the second is to remove objects from the image which are either extremely large or extremely small. Both of these methods together improve the detection speed and accuracy of the algorithm.

4.3.1 Masking/Cropping

One of the most significant noise-reduction methods is to mask or crop the image. Masking simply turns every pixel outside a region-of-interest (ROI) to a value with no meaning (e.g., "0"). This means that the computations in later stages still operate on the full-sized image, but the only useful information lies within the ROI. Cropping, on the other hand, actually reduces the number of pixels to process by discarding pixels outside the ROI. Both of these methods eliminate extra information and speed up the whole algorithm, but the exact amount is dependent on how close the ROI is to the size of the target in the frame. A tight ROI on the target eliminates more excess noise from the frame, and produces fewer pixels to process later on in the algorithm. Although this stage can logically be considered in the "object pruning"

stage, in reality, it can be placed near the beginning of the processing stream in order to achieve maximal benefits from noise elimination and pixel count reduction.

4.3.2 Removing Objects

A process known as *area opening* is used to remove objects ("blobs") with areas less than a given number of pixels from the thresholded image. The general process involves locating blobs by a connected-component labeling algorithm, and then computing the area of each blob in pixels before comparing the result with the input parameter. Blobs with area less than the threshold value are filled with the background value. In order to keep the blobs greater than some value, but smaller than another, two binary images are computed at each threshold value and then multiplied together, after inverting the image produced using the larger threshold value. These threshold values for blob areas are two of the input parameters of our algorithm.

4.4 Marker Detection

The last stage of the algorithm is to locate the marker circles, compute the centers in image coordinates, and reproject the actual marker centers into world coordinates.

4.4.1 Finding Contours

In binary images, contours are curves defined by sets of points along the borders of light or dark regions and enclose connected pixels having the same value. Various algorithms exist to locate contours in images, but a particularly popular one is given in [82], where an input image is scanned from left-to-right and top-to-bottom, and changes from background ("0") to foreground ("1") are recorded as boundaries. Once a boundary is located, a label is assigned and the border is followed until it ends, or the starting point is reached. When the algorithm finishes, a hierarchy of boundaries has been constructed. These contours are then used in the next step as hypotheses for ellipses. Because of the pruning completed in the previous stage, the number of hypotheses generated in this step is significantly less than it would otherwise be.

4.4.2 Fitting Ellipses

This step fits ellipses to the hypotheses generated in the previous step, and in this work we use two variants of the least squares method, which is summarized in Chapter 2.2.1. The first method is Bookstein's method (see Chapter 2.2.1.1 and [57]), and the second is a variant of DLS (see Chapter 2.2.1.3 and [36]). Both of these algorithms rely on solving linear systems of equations, and the results produced by both of these algorithms are similar. These methods return the parameters (center, orientation, and axes length) of the fitted ellipse, which will be used in the next step to filter ellipses.

4.4.3 Ellipse Culling

Using the estimated parameters of the fitted ellipse, the values for a, b, area, arc length, and contour area can be computed. These values are then used to evaluate the ellipse according to a range of expected values, which are passed to the algorithm from a user-configuration file on start-up. The tests include, for example, checking if the arc length of the contour is not too short or long, as well as if the contour area is within reasonable bounds. Similarly, the circularity of the fitted ellipse is checked if it falls within the expected range, and the difference between the ellipse area and the contour area is checked that it does not exceed some maximum. Lastly, the ellipse is checked to ensure it is not too elongated. If a fitted ellipse passes all these tests, then it is considered a marker and added to the list of markers. Next, the detected centers are tested to determine if they form a cluster, and if more than six ellipses are present, the ones outside of the cluster are removed. Once all ellipses have been checked and extra ellipses removed, the list of markers is sorted in two dimensions so that the upper left marker will be first in the list and the lower right marker will be last.

4.4.4 Pose Estimation

Since the ultimate goal of the algorithm is to determine the location of the target in the world coordinate system, it is necessary to know the orientation and position of the camera. The *camera pose* is given by the matrix $[\mathbf{R}|\mathbf{t}]$, where $\mathbf{R}$ is the rotation matrix of the camera

and $\mathbf{t}$ is the position matrix. $\mathbf{R}$ and $\mathbf{t}$ are known as the *external parameters* of the camera. These matrices are used to convert points in the world coordinate system into the camera coordinate system, and vice versa. For example, to convert a 3-D point in world coordinates, call it $\mathbf{M}_w$, into camera coordinates, the equation $\mathbf{M}_c = \mathbf{R}\mathbf{M}_w + \mathbf{t}$ can be used. At this point in the algorithm, the estimated ellipse centers are in camera coordinates, but in order to convert them into world coordinates, it is necessary to compute $\mathbf{R}$ and $\mathbf{t}$. This problem is known as the *camera pose estimation problem*, and has been studied extensively in both the photogrammetry and computer vision domains. To determine the pose of the object relative to the camera, it is necessary to know the location of a set of n points on the object in the coordinate system of the object. In our context, our target has six circles whose center locations are known in the target coordinate system, hence $n = 6$. To solve for $\mathbf{R}$ and $\mathbf{t}$, several methods exist, including Direct Linear Transformation and Perspective-n-Point. Both are used in this work, but due to limited ground truth data, the validity of the results is suspect.

4.4.5 Find Target Location

Once the rotation and translation matrices have been recovered, they can be applied to the marker centers, and then the target position in the world reference frame can be computed. This information can then be used to perform some useful task based on the target position.

CHAPTER 5. METHODOLOGY

When accelerating an algorithm in reconfigurable computing fabric, there is a well known methodology following the basic steps shown in Figure 5.1. First, it is necessary to have at least a basic understanding of the algorithm. Next, before any decisions are made as to which computations of the algorithm to accelerate, it is critical to measure the performance characteristics of the algorithm by profiling. Basically, this step determines empirically which sections of the algorithm to focus on in order to achieve the greatest speed-up. Once profiling results have been collected and analyzed, then design decisions can be made, such as what target platform to use, what components a system utilizing this accelerated algorithm needs, and so on. Next, the architecture of the accelerator can be designed. After the architecture of a component of the algorithm is designed, it is implemented in a hardware description language (HDL), and then any software that the component needs can be written. The last step is to integrate the accelerated component into the system and test the correctness of the algorithm using the accelerated components. This methodology need not be strictly sequential, and many steps can be done in parallel. The rest of the chapter outlines the stages of this process applied to the algorithm described in Chapter 4.

5.1 Algorithmic Understanding

During this phase, the reference Matlab implementation was explored to gain understanding of the algorithm. At the same time, the Matlab code was translated into C++ (a process known as porting) using the OpenCV computer vision library [83]. To ensure a fair comparison of the profiling results between the two implementations, the C++ computations were structured in the same way as in the Matlab version. The similarity of the results between the Matlab and

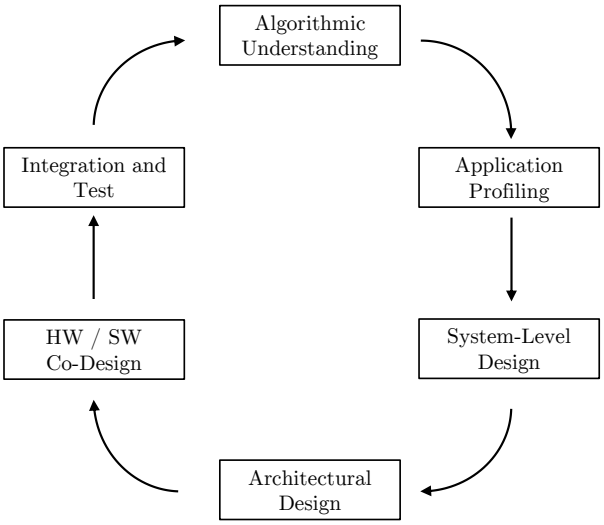


Figure 5.1: The general methodology.

C++ versions was also verified. Once the results were sufficiently close to those from Matlab (within 10^{-6}), the C++ code was considered sufficient to move on to the next step. This initial port is named Version 0.

5.2 Profiling

To determine the performance bottlenecks in the algorithm, both the reference Matlab implementation and Version 0 of the C++ algorithm were profiled. The profiling results from Version 0 were used to perform some optimizations on the C++ algorithm to produce Version 1, which was then re-profiled to see how much improvement was gained.

5.2.1 Matlab

Coarse profiling was performed with the initial Matlab implementation using the built-in tool `profile`. The initial results, shown in Table 5.1, were collected in Matlab running on a 2-GHz x86 machine, and indicate that the `convertImageToBW` (adaptive thresholding) function is the most computationally intensive, consuming 56.1% of the execution time. The next two most computationally complex functions consumed more than 10% of the execution time apiece; `findContours` consumed 13.6% and `findMarkers` 11.9%. Notice that `findContours` extracts the connected groups of pixels from the frame, resulting in a shift from pixel space to contour

Table 5.1: Matlab profiling results, collected with the `profile` tool. Data was collected using 19 images.

Matlab Function	Time (s)	%
convertImagetoBW	6.53	56.1
findContours	1.58	13.6
findMarkers	1.39	11.9
readImage*	0.759	6.5
undistortImageWithParams	0.663	5.7
removeSmallandLargeBlobs	0.398	3.4
findRealWorldPoints	0.150	1.3
saveRotationAndTranslation*	0.077	0.7
convertImageToGrayscale	0.043	0.4
cropImage	0.035	0.3
sortMarkers	0.006	0.1
changeThresholdBlockSize	0.003	0.0
calculateErrors*	0.001	0.0
saveData*	0.001	0.0
overhead*	0.011	0.10
Total	11.6	100.
Avg. FPS	**0.0930**[1]	

[1] Computed value does not include functions marked with *.

space (i.e., the algorithm is no longer processing pixels, but rather contours). `findMarkers` is also performing computations on contours rather than pixels.

The next most expensive function is the one performing disk I/O, and need not be considered for significant optimizations, simply because the final system will not be obtaining images from a disk, but rather a live camera. `undistortImageWithParams` (distortion correction) is next, and it is interesting that it consumes as little time as it does (5.7%), considering it operates on every pixel in the image. The remaining functions consume 4% or less of the the total execution time.

These results provide a reasonable starting point for understanding the computational complexity of the algorithm, but they will not match the C++ version exactly, due to implementation differences between the underlying libraries.

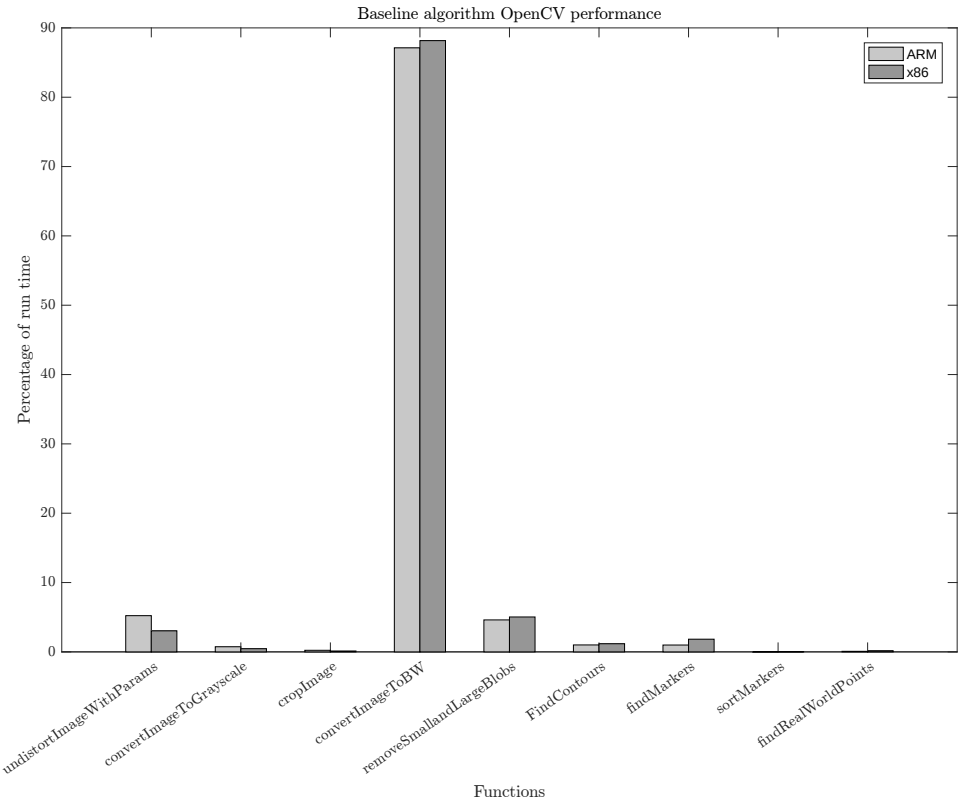


Figure 5.2: Version 0 profiling results using Sauvola's thresholding and no cropping applied to the image.

5.2.2 Version 0

A timing harness is used in Version 0 to measure the time consumed by each stage of the algorithm. To time a portion of code using the harness, two functions are called before and after the block of code under test. `START_STOPWATCH` performs set-up and starts the timer and `STOP_STOPWATCH` stops the timer and computes the time elapsed. These functions use one of the Linux system calls, `gettimeofday` or `clock_gettime`, depending on the compilation flags. `clock_getttime` is accurate to nanoseconds and can use one of the several Linux system clocks. The monotonic system clock was used when profiling the code, which is guaranteed by Linux to not be affected by discontinuous jumps in the system time. Using this harness, the results in Table 5.2 were obtained, and are plotted in Figure 5.2. Results were collected by executing Version 0 on a 2 GHz x86 processor, and on a 666 MHz ARM processor, both running Linux. The algorithm processed a set of 38 frames 10 times to collect timing information, and then the

Table 5.2: Initial profiling results for Version 0. Results are the average times from 10 trials each containing the same 38 test frames. The times for the first full frame in each trial are omitted.

	ARM		x86	
Function	Time (ms)	%	Time (ms)	%
`undistortImageWithParams`	76.1	5.22	2.43	3.04
`convertImageToGrayscale`	10.8	0.741	0.366	0.458
`cropImage`	3.35	0.230	0.103	0.129
`convertImageToBW`	1270	87.1	70.5	88.2
`removeSmallandLargeBlobs`	67.2	4.61	4.03	5.04
`FindContours`	14.6	1.00	0.938	1.17
`findMarkers`	14.4	0.988	1.46	1.83
`sortMarkers`	0.0445	0.00305	0.0119	0.0149
`findRealWorldPoints`	1.18	0.0810	0.129	0.161
Total	1460	100.	80.0	100.
Avg. FPS	**0.685**		**12.5**	

results were averaged for each function, omitting the time contributed by the first frame in each trial. This data represents a best-case performance estimate of the "steady-state" behavior of the algorithm for a particular set of images.

Like those from Matlab, these results show that `convertImageToBW` consumes the most time, 88.2% on x86 and 87.1% on ARM. Interestingly, `undistortImageWithParams` is next highest for the ARM platform, consuming 5.22% of the execution time, but only consuming 3.04% on the x86. The second highest consumer of execution time for x86 is `removeSmallAnd-LargeBlobs`, consuming 5.04%, but only 4.61% on ARM. All of the other functions take much less than 5% of the execution time. The cost of reading images from the disk (magnetic or flash) is not included in these results, since it is a consequence of the early design phase without a live camera stream. The performance for this initial version is less than 1 FPS on ARM, and about 12.5 FPS on x86. Although the x86 platform meets the minimum requirement for the system, it is only used as a reference, whereas the ARM is the target architecture, and is significantly under the minimum 10 FPS, which warrants further effort optimizing the software.

These results indicate the greatest gains in execution time for the ARM platform will come from accelerating the `convertImageToBW` and `undistortImageWithParams` functions,

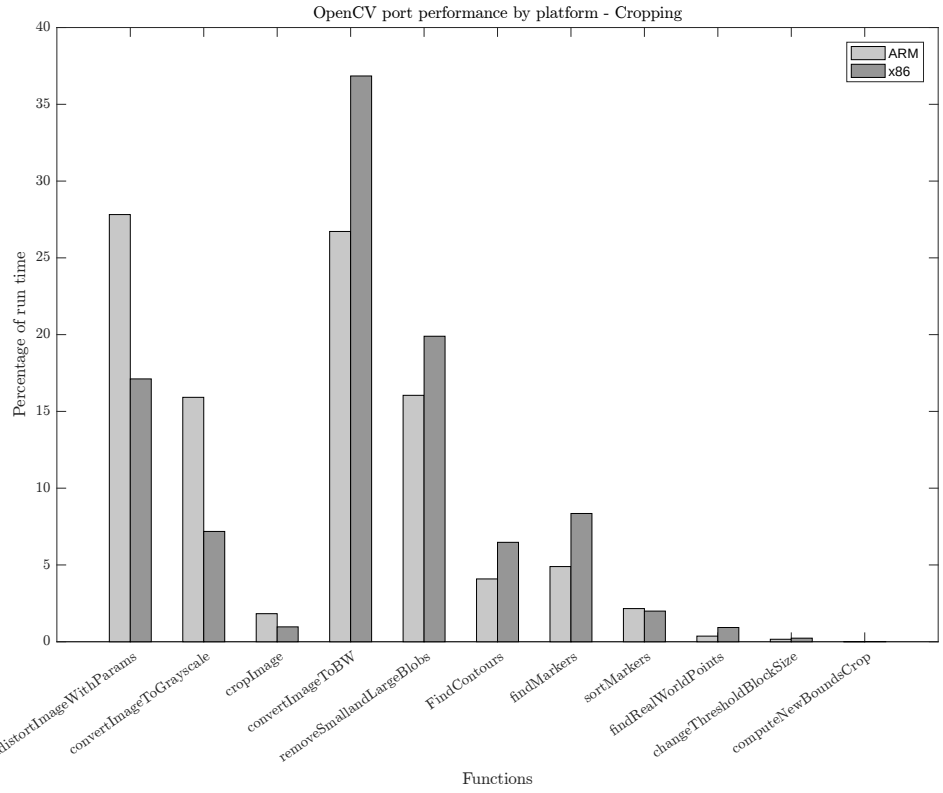


Figure 5.3: Version 1 profiling results using Sauvola's thresholding and applying a region of interest crop to the image.

making these computations potential candidates for hardware acceleration. Both of these functions are well-suited for a streaming pixel pipeline architecture, since they operate on every pixel in a frame, whereas functions from `removeSmallAndLargeBlobs` onward process contours, randomly accessing the necessary image pixels. Using these results, some additional algorithmic optimizations can be made to create a new version of the algorithm, which is labeled Version 1.

5.2.3 Version 1

Two major algorithmic optimizations are made in Version 1. The first optimization is derived from the observations that the target typically occupies a small portion of the frame, and that it typically moves only short distances between frames. Therefore, the logical conclusion is to use the results of the first frame where six circles are successfully located to restrict processing on subsequent frames to the region containing the target in this first frame. Then,

Table 5.3: Profiling results for Version 1 with a ROI crop applied to the frame. Results are the average times from 38 trials each containing the same 10 test frames. The times for the first full frame of each trial are omitted.

	ARM		x86	
Function	Time (ms)	%	Time (ms)	%
`undistortImageWithParams`	52.9	27.8	1.66	17.1
`convertImageToGrayscale`	30.3	15.9	0.696	7.18
`cropImage`	3.47	1.82	0.0938	0.968
`convertImageToBW`	50.8	26.7	3.57	36.8
`removeSmallandLargeBlobs`	30.5	16.0	1.93	19.9
`FindContours`	7.78	4.09	0.628	6.48
`findMarkers`	9.31	4.90	0.810	8.36
`sortMarkers`	4.10	2.16	0.193	1.99
`findRealWorldPoints`	0.694	0.365	0.0898	0.926
`changeThresholdBlockSize`	0.304	0.160	0.0225	0.232
`computeNewBoundsCrop`	0.0116	0.00610	0.00143	0.0148
Total	190.	99.9	9.69	100.
Avg. FPS	**5.26**		**103**	

this region of interest (ROI) can be adjusted after each frame and applied to the next frame to compensate for a moving target. In software, this can be done one of two ways: first, by simply masking the pixels outside the ROI to a value that has no meaning in later computations ("0", for example). However, this method requires iterating over the entire image once to set all pixels outside the ROI to the desired value, and then each subsequent stage may still need to iterate over the whole image. For example, if the thresholded image is masked to a certain ROI, all pixels outside the ROI are set to "0", but in `findContours`, the procedure must still check every pixel in the input image to locate contours. A better method in software is to simply crop the image to the ROI, discarding the extra pixels, which can reduce the number computations in later stages.

Both methods were implemented in the software application, but cropping performed the best, and so was retained. The profiling results using cropping are shown in Figure 5.3, and listed in Table 5.3. As can be seen from the data, cropping before thresholding reduces the percentage of execution time of thresholding on the ARM in Version 1 by about 60%, resulting

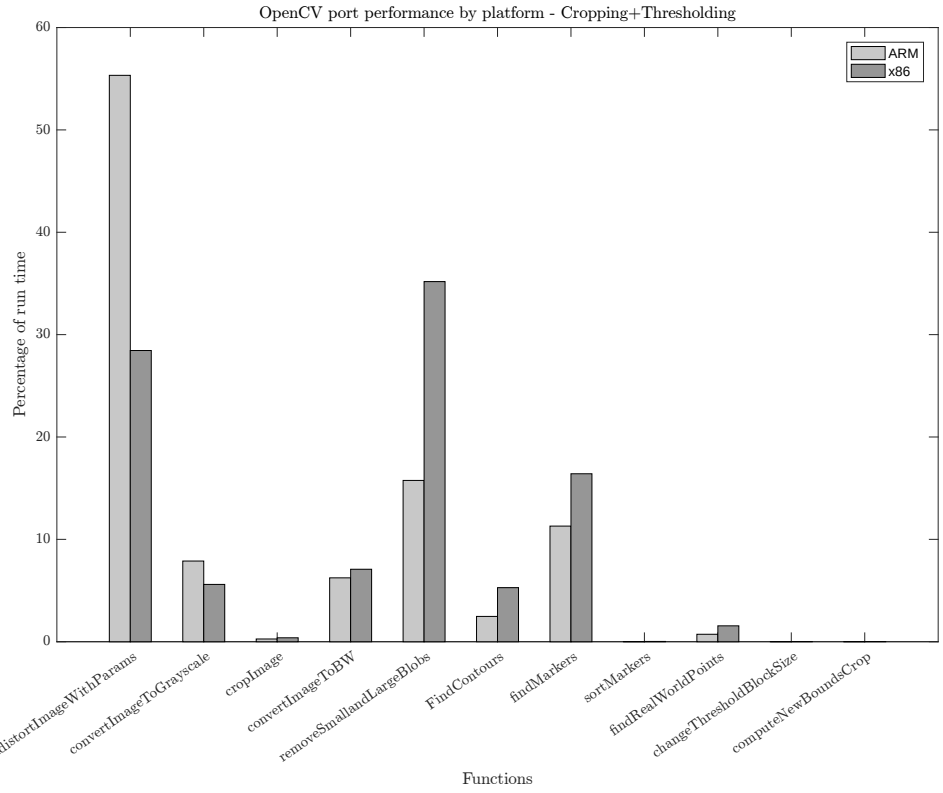


Figure 5.4: Version 1 profiling results using mean thresholding and applying a region of interest crop to the image.

in a 7.7× improvement of the FPS, up to 5.26 FPS. The x86 has a similar performance boost, jumping up to 103 FPS. However, an interesting effect of the CPU data cache size is observable in this new data. As a result of cropping, the most time consuming function on the ARM is now `undistortImageWithParams`. With only 544 KB of data cache memory available on the ARM CPU, a full-size 3 MB image cannot fit entirely in the cache. Therefore, the distortion correction function must wait for higher latency memory accesses at various times during its execution while processing the entire image, significantly impacting performance. Although the speed-up from simply cropping the image is significant, it is still necessary to reduce the execution time of the algorithm to meet the performance requirement.

The second optimization involves alternative means of image thresholding that do not negatively impact detection of the circles on the target. Sauvola's method is undesirable for several reasons: it is slow in software, slightly more computationally complex, and it relies on an integral image as its fundamental performance optimization, which makes multiple passes of a

Table 5.4: Profiling results for Version 1 with a ROI crop applied to the frame and mean thresholding instead of Sauvola's method. Results are the average times from 38 trials each containing the same 10 test frames. The times for the first full frame of each trial are omitted.

	ARM		x86	
Function	Time (ms)	%	Time (ms)	%
`undistortImageWithParams`	75.4	55.3	1.88	28.4
`convertImageToGrayscale`	10.7	7.85	0.371	5.60
`cropImage`	0.364	0.267	0.0255	0.385
`convertImageToBW`	8.50	6.24	0.469	7.08
`removeSmallandLargeBlobs`	21.5	15.8	2.33	35.2
`FindContours`	3.37	2.47	0.350	5.28
`findMarkers`	15.4	11.3	1.09	16.5
`sortMarkers`	0.0133	0.00976	0.00182	0.0275
`findRealWorldPoints`	0.992	0.728	0.103	1.56
`changeThresholdBlockSize`	0.0110	0.00807	0.00123	0.0186
`computeNewBoundsCrop`	0.0117	0.00859	0.00131	0.0198
Total	136	100.	6.62	100.
Avg. FPS	**7.35**		**151**	

sliding window over an image less costly in software. Therefore, Sauvola's method is unsuitable to implement in a pixel-streaming hardware architecture. Fortunately, mean thresholding is another approach, and the results of the algorithm using this method do not deviate significantly from those when using Sauvola's method.

Mean thresholding differs from Sauvola's method in that it simply uses the mean of some local neighborhood around a pixel to determine if a pixel is in the foreground or background, rather than computing the mean and standard deviation of the window. Additionally, no integral image is necessary. After implementing cropping and mean thresholding, the profiling results for Version 1 are shown in Figure 5.4 and listed in Table 5.4. The performance gains on ARM from simply changing the thresholding method are significant, dropping the percentage of execution time spent in `convertImageToBW` from 26.7% after adding cropping to Version 1, to 6.24% in the final implementation. The FPS improved by about 1.4×, to 7.35. x86 again demonstrated a similar performance boost, up to 151 FPS. As can be seen, with only these two software optimizations, we are much closer to our performance target of 10 FPS, but for

any more significant performance gains on the ARM platform, it is necessary to use hardware implementations of certain stages of the algorithm.

5.3 System-level Design

In this stage, the different performance requirements for the algorithm, as well as other non-functional requirements, such as system cost, compatibility with future devices, etc. all must be considered in order to select the final hardware components of the system. In some cases, such as ours, the final system target is simply given. Our three major requirements are to target the Zynq-7000 family of chips from Xilinx, the performance of the final algorithm must be at least 10 FPS, and image distortion correction should be ignored (initially). The Zynq family of chips contain two major components: the processing system (PS), which contains a dual-core ARM processor, and the programmable logic (PL), which is an FPGA. More will be said about the Zynq in Chapter 6.

5.4 Architectural Design

The profiling results from the various software versions determined which stages of the algorithm to accelerate in the PL fabric. An additional requirement is imposed, prohibiting pixels from being streamed out of or into the PS more than once in either direction, since streaming pixels across the PL/PS boundary multiple times negatively impacts performance. Therefore, it is necessary to implement sequential stages of the algorithm as hardware modules until the desired speed-up is achieved. The stages implemented in hardware are grayscale conversion, cropping, and thresholding. Image distortion correction should also be implemented as a hardware module, but is ignored for now, as per the initial requirements. Also note that cropping in a stream of pixels is not trivial and has no performance gain over masking, so the hardware stream simply masks pixels. The design of these three components is described in Chapter 6.

5.5 Hardware/Software Co-design

At this point in the acceleration methodology, it is possible to continue optimizing software while hardware cores are developed in parallel, once a reasonable interface between them is defined. Our interface is described in Chapter 6, and consists of a set of registers used to configure, control, and obtain status from each core.

Testing of the hardware cores was done in two steps – first, by simulation using the Vivado 2015.4 tools, and second, by "bare-metal" testing, where the core was put onto the FPGA, and software written to test the functionality of the core without the complexity of an operating system.

5.6 Integration and Test

The "integrate and test" step is closely tied to the previous, in that as cores become complete enough to integrate into the final system, they are added, one at a time, from upstream to downstream, and software is written to test functionality. If tests fail, then the cause is determined and the core and software revised until functionality is complete. Then the core can be integrated into the final algorithm. Our system design is described in detail in Chapter 6.

5.6.1 Other Testing Infrastructure

An important step of testing is to continuously check that the results from the algorithm are as expected. To facilitate our development and check we were on track, a tool was developed to visualize the results from the algorithm, and is shown in Figure 5.5. The tool, written in Qt 5, displays the original image and a grayscaled image overlaid with the six computed centers as yellow dots. The tool also plots the average FPS (based on timestamp information), instantaneous FPS, the number of centers detected, the x and y positions of the centers over time, a computed confidence value, and the eccentricities of the six ellipses.

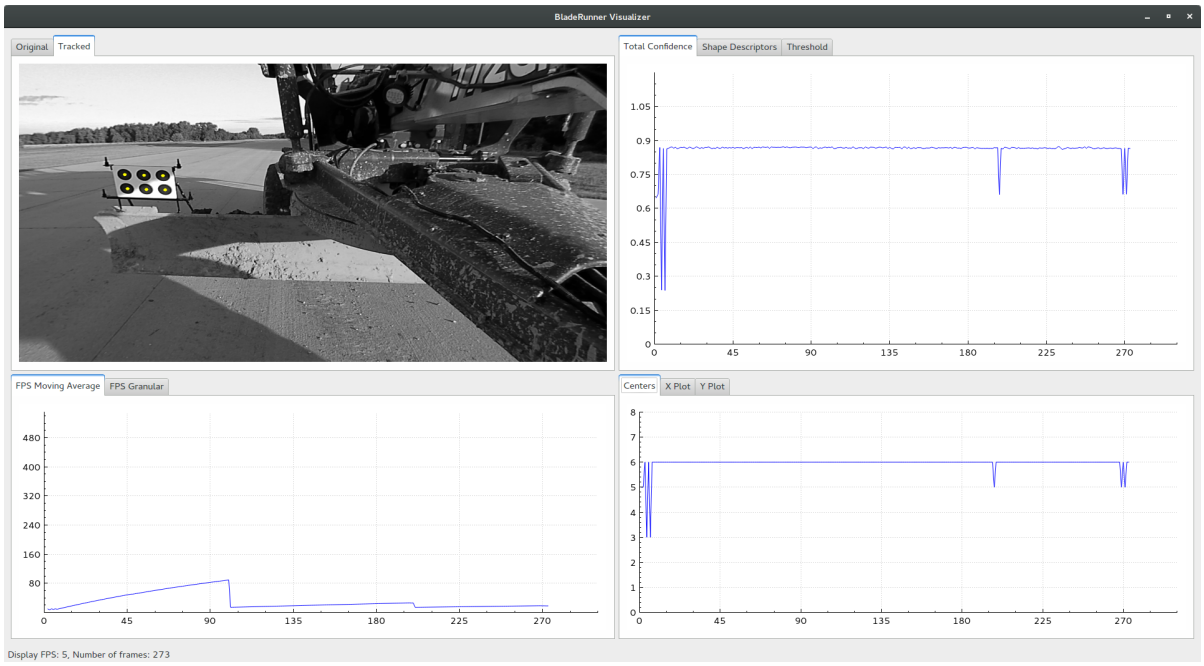


Figure 5.5: A visualization tool to help view and analyze algorithm results.

5.6.2 Regression Testing

Another important sub-stage in this step is to make sure changes to software or hardware do not break anything else. For this application, it is important to detect the same six centers as the Matlab reference implementation every frame. To facilitate this, an automated script was developed that executes the algorithm using user-specified data sets, recording the detected centers, and plotting the number of centers located in each frame. This information can then be used to determine if the implementation changes affect the accuracy of the algorithm in any way.

5.6.3 Summary

This chapter describes the method used to convert a Matlab algorithm implementation, capable of processing images at 0.0930 FPS, into an OpenCV software application 79× faster on our target ARM platform. The next chapter describes the hardware/software architecture which finally achieves the necessary performance gains to fulfill our design requirements.

CHAPTER 6. ARCHITECTURE

In this chapter, the system architecture is described, including the target platform, hardware architecture, and software architecture.

6.1 System

The target platform for this work is an Avnet ZedBoard containing a Zynq 7020 SoC. The architecture of the Zynq is shown in Figure 6.1. As the diagram shows, the chip contains a dual-core ARM Cortex A-9 Application Processor Unit (APU), as well as the necessary components to communicate with the programmable logic (PL) and other off-chip peripherals. The ZedBoard also contains 512 MB of DDR3 memory clocked at 533 MHz, and an SD card interface. The ARM APU operates at a frequency of 666 MHz.

Because the software algorithm depends heavily on the OpenCV computer vision library [83], it is necessary to run an operating system (OS) on the Zynq chip. An ARM Arch Linux distribution was selected (see Table 6.1 for version information) for its small size and software tool chain. These tools enabled native compilation of the software on the ARM processor. The main tools and libraries used are listed in Table 6.1.

Table 6.1: Software library version information.

Name	Version
Arch Linux	3.19.0-3
U-Boot	2016.01-03961
GCC	5.3.0
OpenCV	3.1.0

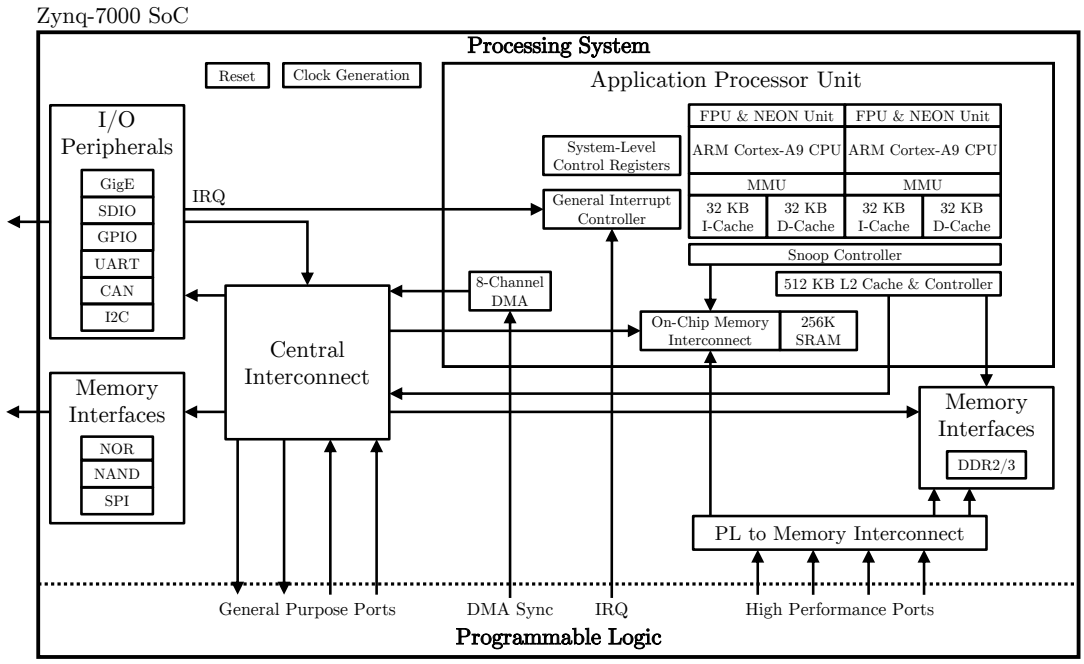


Figure 6.1: Zynq 7000 SoC (adapted from pg. 27 in [84]).

6.2 Booting the Zynq

The Zynq boot process is important for system configuration and initialization. The system is configured to follow the unsecured boot steps using an SD card. The general steps of this process are: first, the BootROM executes on CPU0 and CPU1 simultaneously to determine the CPU ordering. Once the ordering is determined, CPU1 executes a `wait for event` instruction, while CPU0 continues to execute the BootROM.

On CPU0, the BootROM checks the MIO pin settings to determine the boot device. Because the system is configured to boot from an SD card, the BootROM reads the BootROM header contained in the BOOT.BIN image stored on the SD card, loads the image into on-chip memory, and transfers execution to the First Stage Bootloader (FSBL) contained in the BOOT.BIN file. The FSBL uses the PS7 configuration information generated by the Xilinx tools to finish initializing the PS, which includes DDR memory and other I/O settings, and then loads the bitstream into the PL. Once programming the PL completes, the FSBL loads the second stage bootloader into DDR memory and transitions control to it.

Table 6.2: Modified U-Boot variables.

Name	Value
`kernel_size`	`0x500000`
`ramdisk_size`	`0x5E0000`
`bootargs`	`mem=496M`
`fdt_high`	`0x1F000000`
`initrd_high`	`0x1F000000`

The second stage bootloader is the U-Boot [85] program, which handles the rest of the system configuration before handing final system control off to the OS. U-Boot operates by running commands with configurable parameters. These commands and parameters can be stored in variables in the U-Boot environment. The command to boot the OS is stored in `bootcmd`, which is automatically launched after a configurable delay.

Our design requires the ability to read and write to a section of RAM not controlled by the OS. To that end, the default values for the `bootargs`, `fdt_high`, and `initrd_high` variables were modified to only allow the lower 496 MB of RAM to be used by the Linux OS, thereby reserving the upper 16 MB for use by the hardware pipeline. Table 6.2 shows how the important U-Boot variables are modified from their default values to facilitate this configuration. The last task performed by U-Boot is to load the OS into memory (typically as a RAM disk image) and then launch it using the command in the `bootcmd` variable. This concludes the boot phase of system start-up. More details can be found in [84].

6.3 Accelerator Architecture

Once the system is properly configured and the OS is loaded, the accelerator has the high-level architecture depicted in Figure 6.2. The architecture contains two different computational model types, roughly delineated by the dotted line separating the PS from the PL in the diagram. The PS uses a sequential computational model, where groups of points are operated on in units of frames rather than on the level of individual pixels. The PL, however, operates on each pixel as it travels through the pipeline, and is known as a "streaming" architecture.

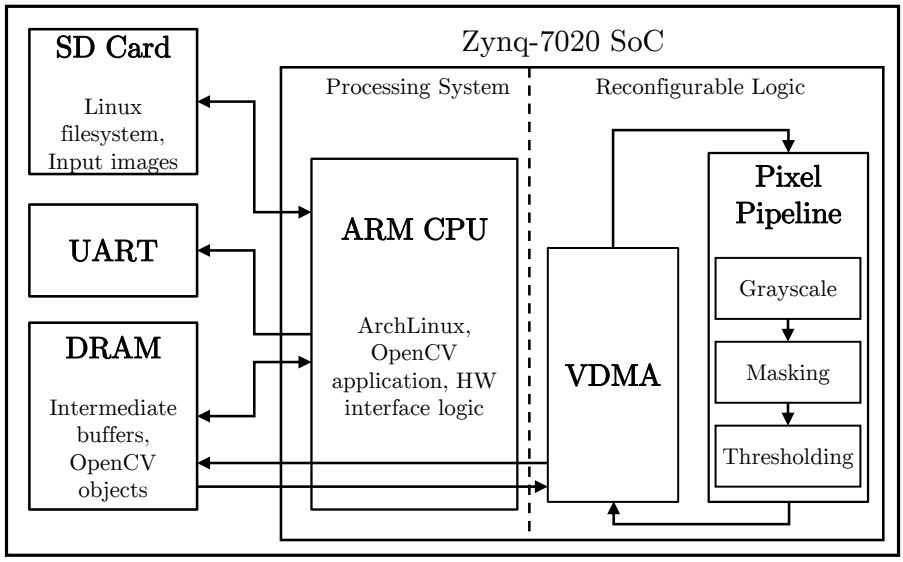


Figure 6.2: The system architecture.

To convert from the sequential frame-based software architecture to the pixel stream architecture, a Video Direct Memory Access (VDMA) intellectual property (IP) core provided by Xilinx [86] is used. This core reads from a location in DRAM memory and, when configured with the pixel size information, outputs a pixel into the pipeline every clock cycle. Similarly, when the stream processing is finished, each pixel is written back to a DRAM frame buffer, where the software application can access it.

There are several advantages to stream processing an image rather than processing using a sequential CPU architecture. The most significant advantage is limiting the reads and writes from DRAM, which is very slow. This fact, coupled with the observation that most of the pixel computations do not require global knowledge of other pixels, that is, operations are typically independent, or only require knowledge within a neighborhood of pixels, provides a strong argument for a stream architecture for the pixel-to-pixel operations. Once meaningful data from the frame is extracted, sequential processing on the ARM APU again becomes feasible in real time, because there is a significant reduction in the amount of data to process. Because of the faster clock frequency of the PS and the reduced amount of data, it is not necessary to implement the entire algorithm in the PL. Rather, functions from the OpenCV library can be used to finish extracting the appropriate data from the frame.

6.3.1 Hardware Architecture

Our hardware architecture, as described in this section, consists of three IP cores. Each core has an AXI Stream (AXI-S) interface, as well as an AXI bus interface, and 16 registers that can be read or written by software applications in order to provide a standardized interface for control and status information. The three IP cores outlined in this section are a grayscale conversion core, a masking core, and an adaptive thresholding core. A fourth core, briefly described before these three, is a Video Direct Memory Access (VDMA) core provided by Xilinx, and is necessary for the design as outlined above.

6.3.2 AXI-S Wrapper

Because of the streaming computation model of the design, a communication protocol between cores is necessary. Xilinx favors AXI4, so our cores were designed to have AXI4-S interfaces. Each core has both a master and a slave interface, and each slave interface is connected directly to the master interface of its immediate upstream neighbor, creating a "daisy chain" pipeline. The AXI4-S protocol [87] defines nine signals: `TVALID`, `TREADY`, `TDATA`, `TSTRB`, `TKEEP`, `TLAST`, `TID`, `TDEST`, and `TUSER`. The `TVALID` and `TREADY` signals determine when the slave reads data from the `TDATA` input. Each core also has FIFOs on the input and output interfaces to avoid pipeline stalls and facilitate a continuous stream of pixels.

6.3.3 AXI Wrapper

In order to access the cores from software, which is not the purpose of the AXI4-S interface, it is necessary to also add an interface that allows memory mapping. The AXI4 protocol [88] allows a slave interface to be written to and read from by one or more master interfaces. The AXI4 interface has four channels that allow data to flow simultaneously between a master and a slave. When connected to an interconnect, a single master can read and write to multiple slaves. In our system, the PS is connected to the hardware cores using an AXI interconnect, enabling read and write functionality.

Table 6.3: Default IP core register configuration.

Register	Default Range	Bits Used	Use
0	0-3	0-1	Control/Reset
1	0-3	0-1	Global Status
2	0	None	Global Error
3	$0\text{-}2^{32}-1$	0-31	Horizontal Size
4	$0\text{-}2^{32}-1$	0-31	Vertical Size
5	$0\text{-}2^{32}-1$	0-31	Frames Processed
6	$0\text{-}2^{32}-1$	0-31	Current Column Offset
7	$0\text{-}2^{32}-1$	0-31	Current Row Offset
8-11	$0\text{-}2^{32}-1$	0-31	Core-specific Config/Parameters
12-15	$0\text{-}2^{32}-1$	0-31	Core-specific Status

6.3.4 Register Interface

The cores are designed with a synthesis-time customizable number of registers, the default number of which is 16. The register width is also customizable, but this design uses 32-bit wide registers. Table 6.3 shows the general structure of the register layout. Register 0 is for control and reset. Bit 0 of this register is a software reset for the core. Setting this bit to "1" will reset the core and hold it in reset until a "0" is written. Bit 1 of register 0 is the enable bit for the core ("1" enables the core, "0" disables it). When a core is disabled, it is considered to be in a "pass-through" mode, where the input pixels will be sent to the output unmodified. Register 1 is a status register, which displays the state of bits 0 and 1 of Register 0 by default, but can be modified to provide more status information for more complex cores. Register 2 is an error register and each core can define the meaning of the bits. Registers 3 and 4 are written by the software to inform the core of the horizontal and vertical frame dimensions, respectively, in pixels.

Each of the custom streaming cores also has a built-in mechanism to keep track of the number of frames processed since the last reset. This value is stored in Register 5, and is incremented whenever the last pixel in a frame is reached. The current pixel in the pipeline can be monitored through Registers 6 and 7, which are read-only registers that provide the column and row offset, respectively, of this pixel. Registers 8-11 are intended to be used as

core-specific configuration or parameters registers, and registers 12-15 are intended to be used as core-specific status registers.

6.3.5 VDMA IP Core

As mentioned above, in order to read frames out of the source frame buffer in DRAM and then write them back into the sink frame buffer in DRAM (a different location), it is necessary to use a core that converts the frame buffer to a stream, and from a stream back to a frame buffer. Xilinx provides the VDMA core [86] to do this. There are multiple streaming ports on the core to allow the same VDMA to both stream pixels from memory (the stream side of the memory-to-stream interface is an AXI-S master) and return the stream to memory (the stream side of the stream-to-memory interface is an AXI-S slave). In order for this core to correctly operate, it needs to be pointed to the physical addresses of DRAM for reading and writing. For a bare-metal application (no OS), this is straightforward, but when using a modern operating system with virtual memory this becomes problematic. There are several different ways around this issue, including writing a kernel driver, mapping a user space buffer, or partitioning the RAM. The last option is used in this system, and an upper limit of 496 MB of RAM is imposed on the OS. The remaining 16 MB of RAM are reserved for use by the VDMA. The software application is then able to read and write to these logical frame buffers using the `mmap` system call, and the VDMA is able to read and write without conflicting with the OS. The initial frame buffer-to-stream interface of the VDMA is an artifact of our development configuration, lacking a live camera as the image source, so this portion of the design will not be present in a full system.

6.3.6 Grayscale Conversion IP Core

The first core following the VDMA in our hardware pipeline implements a standard RGB to grayscale conversion function. The equation for the conversion is

$$Y = 0.299R + 0.587G + 0.114B$$

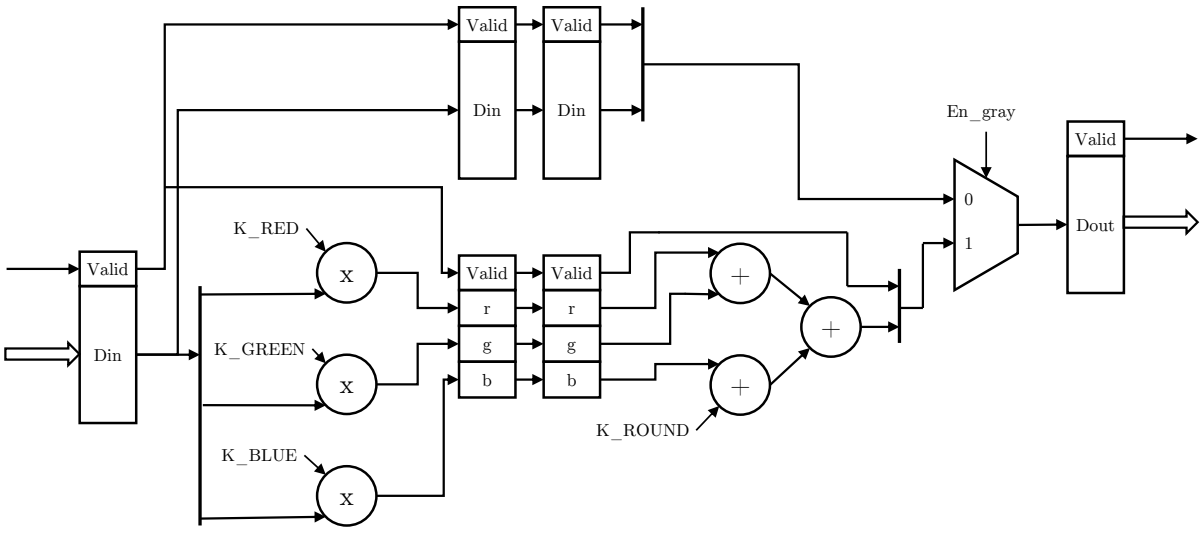


Figure 6.3: The grayscale conversion architecture.

where the values of R, G, and B correspond to the values of the red, green, and blue channels of the input pixel, respectively. Y is the output intensity value in the range $[0, 255]$. In order to avoid floating point operations, costly in both computation time and DSP resources, we used $Q2.14$ fixed point notation for the computation, where 2 is the number of bits to represent the integer component (left of the radix) and 14 is the number of bits to represent the fractional component (right of the radix). These numbers in our design are a synthesis-time parameter. Using fixed point computations, the above equation becomes

$$Y = (4899)R + (9617)G + (1868)B + K\_ROUND$$

where $K\_ROUND = 1 << (fractionbits - 1) = 8192$ in this case. K_ROUND improves the accuracy of the multiplication result by rounding the final result in the appropriate direction.

Because the multiplication could result in a 24-bit value (coefficients are 16 bits, channels are 8 bits), the intermediate multiplications are stored in 24-bit registers and the summation also uses these 24-bit coefficients. The final 8-bit result is obtained by right shifting the 24-bit sum by the number of fraction bits, thus completing the computation. Figure 6.3 shows the grayscale conversion core architecture. The final shift is not shown, since in VHDL, it is simply a matter of routing the appropriate wires to the output. Additionally, the second register after the multiplication is included to allow the tools to execute some re-timing optimizations.

Table 6.4: Grayscale conversion IP core custom register map.

Register	Range	Bits Used	Use
12	0-$2^{16}-1$	0-15	Red Coefficient
13	0-$2^{16}-1$	0-15	Green Coefficient
14	0-$2^{16}-1$	0-15	Blue Coefficient
15	0-$2^{16}-1$	0-15	K

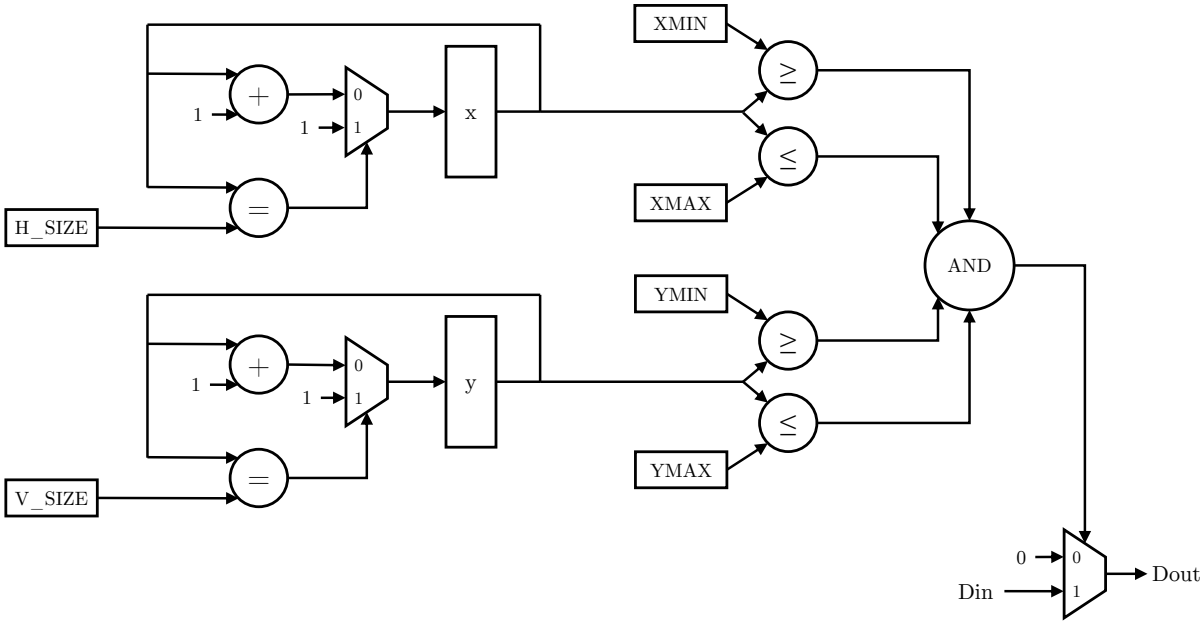


Figure 6.4: The masking architecture.

Finally, a different grayscale conversion core may or may not be necessary in a system with an actual camera, depending on the output obtained from the image sensor.

The register map for the core is shown in Table 6.4. The core-customizable status registers are used to provide an application with the values of the coefficients and rounding constant for debugging.

6.3.7 Masking IP Core

The masking core, depicted in Figure 6.4, is very simple. If the coordinate of the current pixel satisfies the region-of-interest (ROI) condition, the input pixel is transferred to the output unmodified, but if the condition is not satisfied, the pixel value is cleared to 0. The ROI

condition is

$$(x_{min} \leq x_{cur} \leq x_{max}) \wedge (y_{min} \leq y_{cur} \leq y_{max})$$

It is important to note that these inequalities include the pixels on the boundaries of the ROI in the output region. Additionally, this core does not modify the default set of control and status registers.

6.3.8 Adaptive Thresholding IP Core

The core depicted in Figure 6.5 implements an adaptive thresholding computation, which simply converts an entire frame to a binary image, where a pixel has a value of "1" if it is considered to be in the foreground, and "0" if it is considered as part of the background. If the input pixel has a value greater than the mean value of some neighborhood of pixels surrounding the input pixel, it is considered part of the foreground. There are two parameters that make the core "adaptive": the window (neighborhood) size (k), up to some fixed maximum (K), and a constant offset (C), which can be used to adjust for different illumination conditions. The thresholding condition can be expressed as

$$\frac{\sum_{y-\frac{k}{2} \leq j \leq y+\frac{k}{2}} \sum_{x-\frac{k}{2} \leq i \leq x+\frac{k}{2}} p_{ij}}{k^2} - C > P(x,y)$$

where x is the current pixel column position, y is the current pixel row position, and $P(x,y)$ is the intensity value at that location. To avoid the division operation, which is expensive in hardware, the inequality can be rewritten as

$$\sum_{y-\frac{k}{2} \leq j \leq y+\frac{k}{2}} \sum_{x-\frac{k}{2} \leq i \leq x+\frac{k}{2}} p_{ij} > k^2(P(x,y) + C)$$

When this inequality evaluates to `true`, the output pixel is "1", otherwise the output pixel is "0". In order to keep the input pixel as the true center of the neighborhood, the design requires both k and K be odd.

There are several ways to handle pixels on the image borders with a filter such as this, and the methods selected for this core are as follows: for the top border of the image, "0"-valued pixels are output until enough pixels have been buffered to fill a complete window. Once there are enough pixels to fill a window, the thresholded pixel values are output until the window is

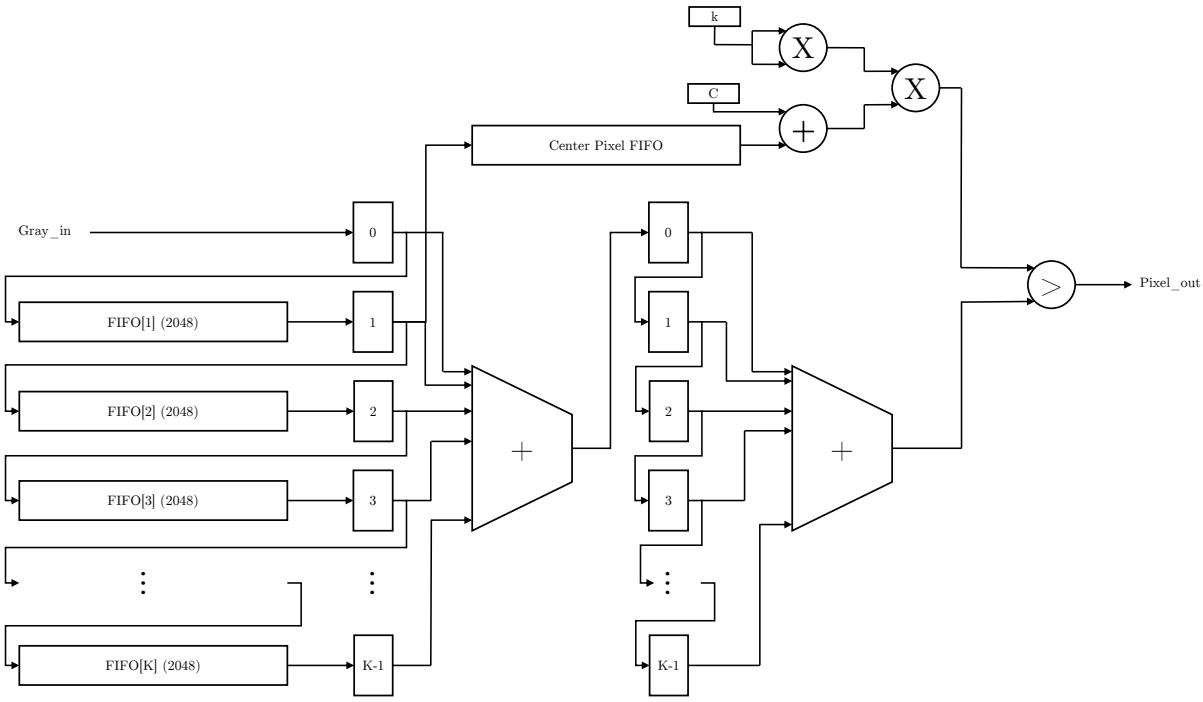


Figure 6.5: The adaptive threshold architecture.

no longer able to be kept full, at which point the core again outputs "0"-valued pixels. This translates to black borders on the top and bottom of the output frame of size

$$\texttt{TOP BORDER SIZE} = (\text{H\_SIZE} + 1) * \left\lfloor \frac{k}{2} \right\rfloor$$

This means the first center pixel is the

$$(k-1) * \text{H\_SIZE} - \left(\text{H\_SIZE} - \left\lfloor \frac{k}{2} \right\rfloor\right)$$

pixel to enter the pipeline and the last center is the

$$\text{V\_SIZE} * \text{H\_SIZE} - \left((\text{H\_SIZE} + 1) * \left\lfloor \frac{k}{2} \right\rfloor\right)$$

pixel to enter the pipeline. The edges of the frame are handled by wrapping around. That is, as pixels are streamed into the core, traversing the frame left-to-right, pixel centers on the left edge will have pixels in the window from the right side of the image, and pixel centers on the right edge will have pixels from the left side of the image in the window. This may produce some interesting artifacts on these edges of the frame, but as long as the circles of the target are not at the very edge of the frame, there should be no adverse effects.

6.3.8.1 Design Features: FIFOs

As shown in the figure, there are K pixel FIFOs that are 2048 pixels deep and 8 bits wide, and one center pixel FIFO that is 131,072 pixels deep and 8 bits wide[1]. The line buffers only need to be as deep as the horizontal size of the image frame, so this design can support frames up to 2048 pixels wide. The center pixel FIFO has as input the registered output from the first line FIFO (register 1 in the diagram). However, the first center pixel is not written to the FIFO until the first $k \times k$ window is completely filled, but once this happens, then each subsequent pixel is added to the center pixel FIFO until the point when the window can no longer be filled with a new pixel value. When a new pixel enters the core, it is buffered for a cycle, then is used as a coefficient in the column sum, and enters the next line buffer. This pattern continues until the pixel finally leaves the sliding window range completely.

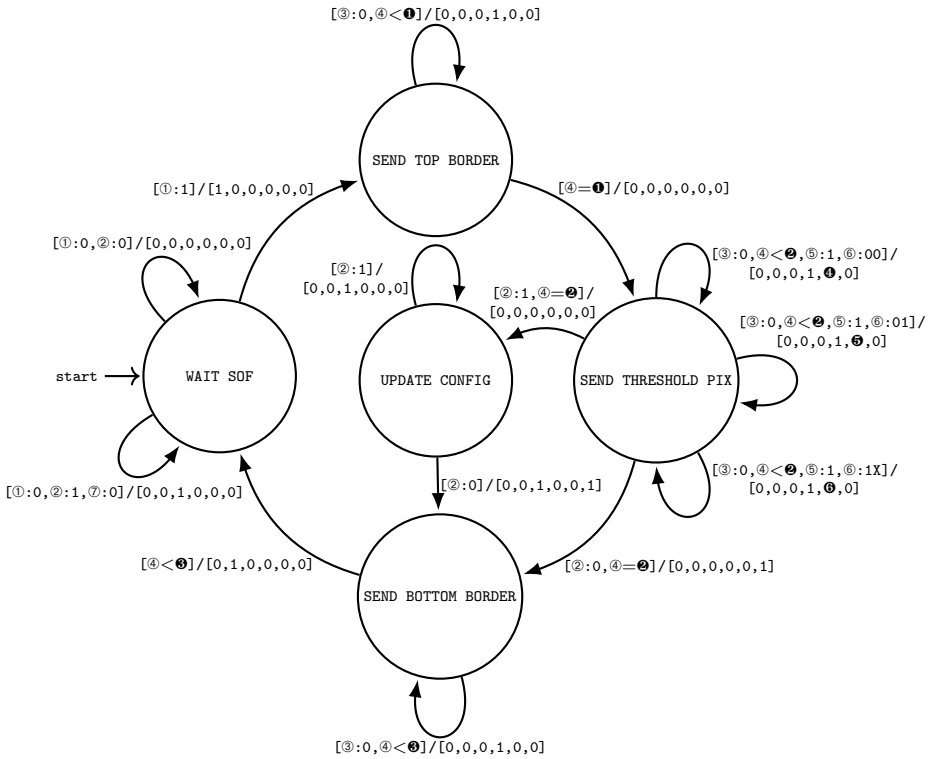
6.3.8.2 Design Features: Adder Trees

The two adder trees compute the sum of the pixels in a given window. The first adder tree computes the column sum, and the second adder tree computes the sum of the column sums. The adder trees are pipelined such that an addition takes place every clock cycle, and the final result is available after $\lceil \log_2 k \rceil$ cycles. The column sums produced by the first adder tree are placed into a shift register to be used again as the window slides across the frame. The output from the second adder is synchronized with the result of the $k^2(P(x,y) + C)$ computation.

6.3.8.3 Design Features: Control Logic

Although much of the internal control logic is managed simply by synchronization of computations via pipelining, there is also a state machine that manages certain events to allow the core to function correctly and produce the right outputs at the right time. When the core first comes out of reset, it is in the `WAIT SOF` state, which merely waits until a start-of-frame (SOF) is received at the input. The software-configurable registers are also able to be written in this state, without transitioning to the `UPDATE CONFIG` state. Once the SOF has been received, the

[1]Note that this center pixel FIFO does not need to be this large, and a future version of the core will reduce this FIFO to a more reasonable size.



Input value vector:[❶:TOP BORDER SIZE,❷:MAX THRESH INDEX,❸:IMAGE SIZE,❹:thresholded value,❺:window sum,❻:window center value]
Input signal vector:[①:sof,②:config pending,③:output buffer full,④:pixel count,⑤:write enable,⑥:output select,⑦:FIFO1 num read]
Output vector:[reset sof,reset pixel count,update config,pixel valid,pixel value,reset FIFOs]

Figure 6.6: Adaptive threshold control state machine.

core transitions into the `SEND TOP BORDER` state, where `TOP BORDER SIZE` "0"-valued pixels are sent to the output. During this time, the core is also reading pixels from the upstream core to fill the first window. Once all of the top border pixels have been written, the core moves into the `SEND THRESHOLD PIX` state, where it outputs thresholded pixels until all thresholded pixels have been computed and sent downstream. This condition is met when

$$\texttt{MAX THRESH INDEX} = \text{V\_SIZE} * \text{H\_SIZE} - \left((\text{H\_SIZE} + 1) * \left\lfloor \frac{k}{2} \right\rfloor \right)$$

pixels have been processed. At this point, the core can either transition to the final `SEND BOTTOM BORDER` state if there are no pending configuration changes, e.g., a new window size (k), or, if there are, to the `UPDATE CONFIG` state. If the core moves into the `UPDATE CONFIG` state, it will propagate changes to the appropriate portions of the core, resetting parts where appropriate, and when finished, will transition to the `SEND BOTTOM BORDER` state. In this final state, the core will write "0"-valued pixels to the output until the output pixel count reaches the full image resolution, that is H_SIZE $*$ V_SIZE pixels. Then, the core returns to the `WAIT SOF` state after resetting the output pixel count to 0. This whole process is shown in the state machine in Figure 6.6, where the input signals prompting a transition are shown on the left of the "/" and the output signals resulting from the inputs are shown on the right of the "/." The meaning of the signals is noted at the bottom of the figure. Computed values which are used for comparisons or as outputs are noted in the `input signal vector`.

6.3.8.4 Design Features: Configuration Registers

Table 6.5 shows the customization of the standard core configuration and status registers. The first eight registers remain unchanged. Four bits of Register 8 are used: bit 0 is for an internal application reset, which is currently unused. Bits 1 and 2 configure the output of the core. Typical usage in a system with the full hardware pipeline would leave these bits set to "00" to produce the expected thresholded output, but when developing the core in a bare-metal environment, the other outputs are useful for debugging. Other output options include the sums ("01") and the pixel center values ("10"). Bit 3 is very important, as it notifies the core to load new parameters. To properly load new parameters to the core, the application

Table 6.5: Threshold IP core custom register map.

Register	Range	Bits Used	Use
8	0-15	0-3	Internal Reset/Configuration/Load Parameters
9	0-$2^{32}-1$	0-31	k
10	0-$2^{32}-1$	0-7	C
11	0-$2^{32}-1$	None	Reserved
12	0-$2^{32}-1$	0-31	Core Version
13	0-$2^{32}-1$	0-31	Current k
14	0-$2^{32}-1$	0-31	Current C
15	0-$2^{32}-1$	0-31	K

must write new values to the desired registers, set this bit to "1," and then set the value back to "0" to notify the core of configuration changes and schedule an update at the next available opportunity, usually after the current frame is fully processed.

Register 9 allows an application to update the window size (k), up to the synthesis-time defined maximum window size (K). Register 10 allows the application to update the constant offset value (C). Even though the core allows a 32-bit value to be written to this register, only the lower 8 bits will be used in the computation.

To aid debugging and integration of the core, Register 12 is customized to serve as a version register, holding the current version number of the core. Register 13 holds the current window size (k), Register 14 holds the current constant offset value (C), and Register 15 holds the maximum supported window size (K) of the core.

6.3.9 Embedded Software

To use these three cores once they are fully integrated into the system, the software algorithm must read an image, write it to the frame buffer from which the VDMA reads data (the *source*), wait until the hardware pipeline finishes processing the frame, and then read back the finished frame from the destination frame buffer (the *sink*), before resuming processing with contour extraction. An abstract C++ class is used to provide a consistent interface between the hardware and software components, implementing common set up and tear down functions for the hardware cores, and also declaring virtual functions for implementation in derived classes

Table 6.6: Hardware abstraction layer application program interface.

Function	Type
`init`	Implemented
`teardown`	Implemented
`write_register`	Implemented
`read_register`	Implemented
`setbits_in_register`	Implemented
`clearbits_in_register`	Implemented
`start`	Virtual
`halt`	Virtual
`reset`	Virtual
`print_registers`	Virtual
`waitFor`	Virtual

to account for core-specific behavior. The functions in the base class are listed in Table 6.6, and denoted as implemented or virtual.

The `init` and `teardown` functions use `mmap` and `munmap`, respectively, to obtain and clean up pointers to the configuration registers for each core. `write_register` and `read_register` write 32-bit values to and read 32-bit values from a given core's registers, while `setbits_in_register` and `clearbits_in_register` allow manipulation of individual bits in these registers.

The virtual functions implemented in each derived class are `start`, which enables the core to begin processing pixels, `halt`, to stop processing, passing pixels through the core unmodified, `reset`, to reset the core, `print_registers`, to read all the register values from a core and write them to the standard output, and `waitFor`, a function that waits until the frame count register of a core reaches or exceeds the input value. This last function is allowed to be implemented on a per-core basis, but a common implementation method is as a spin-lock on the frame count register.

Standardizing the interface to the hardware is a desirable goal, but difficult to implement well. In practice, our individual cores had enough differences to make deriving classes from this base class tricky, particularly in the case of the VDMA interface. This means, however, opportunity exists to create a more robust hardware/software interface in future work.

CHAPTER 7. RESULTS

This chapter describes the accuracy, resource utilization, and performance of our implementation.

7.1 Accuracy

For this work, since we did not have any reliable ground truth data, our accuracy metric was simply finding the correct number of ellipses (six) that fit the constraints of a circle on the target, and then a visual inspection of the centers to verify they were approximately the centers of the markers in the input image. Both of our OpenCV versions of the algorithm located the same six centers found by the Matlab reference implementation, and in some cases found six marker centers when the Matlab algorithm located fewer.

7.2 Resource Utilization

Table 7.1 shows the post-synthesis resource utilization of the design when targeting the Zynq 7z020 SoC and using a maximum window size (K) of 101. This value was empirically chosen by testing our software algorithm with our data sets and allowing the threshold window size to grow unbounded, and then observing that the window size rarely exceeded 101. As the table shows, the resource utilization at this design point consumes little reprogrammable fabric, only 27.56% of LUTs, 1.78% of LUTRAMs, and 18.86% of FFs are used, but uses 64.64% of available BRAMs. The Vivado 2015.4 tool chain from Xilinx was used to generate the results.

Figure 7.1 depicts the resource utilization for values of K in the range $[11, 191]$ with an interval of 30. The plot shows the resource utilization of the design scales linearly for nearly all of the resource types, and the maximum window size of the thresholding core is near 191,

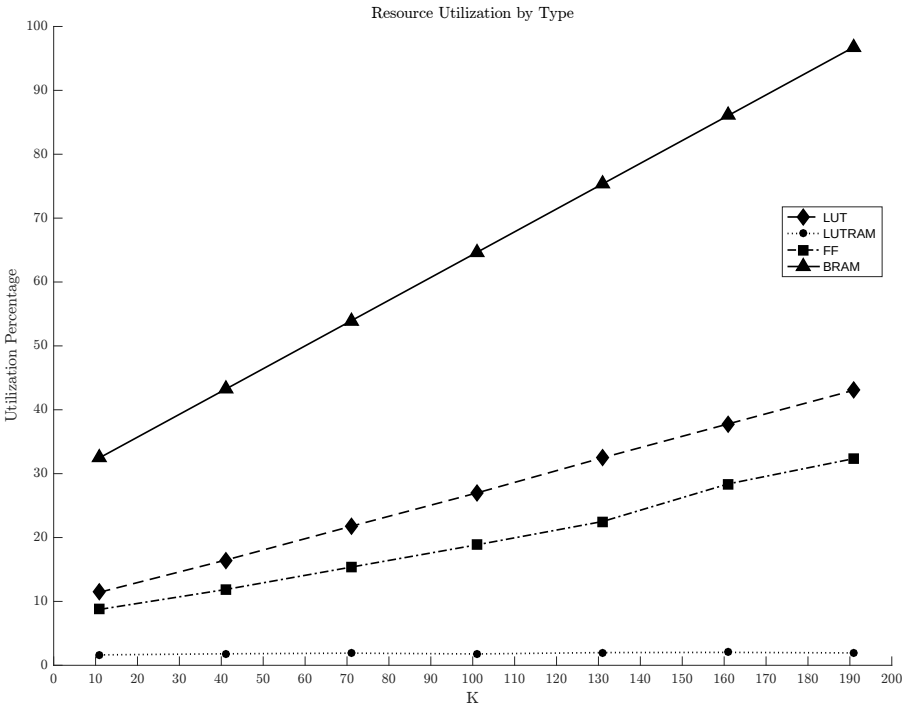


Figure 7.1: Post-synthesis resource utilization for different values of K.

but cannot grow much larger due to the limited number of device BRAMs. For example, when $K = 191$, 96.79% of the BRAMs are used. Further optimization of the thresholding core could improve the upper bound on this device. Additionally, the utilization of device DSPs, IO pins, and BUFG resources is constant for these design points. Note that this version of the design does not include the image distortion correction hardware core.

Table 7.1: Resource utilization for a design with $K = 101$ on the Zynq 7z020.

Resource	Utilization	Available	% used
LUT	14664	53200	27.56
LUTRAM	310	17400	1.78
FF	20066	106400	18.86
BRAM	90.5	140	64.64
DSP	8	220	3.64
IO	21	200	10.50
BUFG	1	32	3.12

Table 7.2: Average FPS for the initial software algorithms. See Chapter 5 for more details.

Platform	FPS
MATLAB	0.0930
ARM	0.685
x86	12.5

7.3 Performance

Our primary performance measurement is in frames per second (FPS), and our system is required to achieve a minimum of 10 FPS on the Zynq in order to be considered successful. Our initial profiling resulted in the FPS numbers shown in Table 7.2. As can be seen, the FPS of all implementations were poor, but were especially bad for the Matlab implementation, which only processed 0.0930 FPS. On the ARM platform, the algorithm did better, processing 0.685 FPS, while the best performance was on the x86 architecture, where 12.5 FPS were processed. The algorithm which generated these results includes image distortion correction in software, and does not implement cropping, and as a result, is the worst-case performance.

In the following discussion, Table 7.3 shows the average FPS for different hardware/software configurations on the Zynq SoC compared with Matlab implementations. Table 7.4 gives the configuration associated with each name, which is descriptive of the data set and design parameters used. For instance, a hardware configuration name can include a "G," which means grayscale conversion is included, an "M" for masking, and a "T" for thresholding. A software configuration with "C" in the name means ROI cropping is applied to the images, and if absent, the full frame is always processed. The FPS data for the different configurations are plotted in Figure 7.2. In the figure, the FPS value is represented by the total height of the bar, and the subsections of each bar represent the proportion of the total time spent in the various stages of the algorithm. All of the results were collected using previously distortion-corrected images, due to technical limitations of the ZedBoard, from two data sets. The first, *moving*, contains images where the target and camera are attached to a moving vehicle. A rectangular ROI containing the target occupies $\approx 4,900$ pixels in the upper-left quadrant of the frame, varying very little over the course of the data set. The second data set, used only for the *sw-facing-C*

Table 7.3: FPS results for various hardware/software configurations compared with the Matlab reference implementation (middle column) and the Matlab implementation corresponding to the Version 1 algorithm (last column).

Configuration	FPS	Speed-up (ref)	Speed-up (v1)
matlab-ref	0.093	1.0	–
matlab-v1	56	602	1.0
hw-moving-GMT	64	688	1.14
sw-moving-C	61	656	1.09
hw-moving-GM	60	645	1.07
hw-moving-MT	38	409	0.68
sw-facing-C	31	333	0.55
sw-moving	3	32	0.05
hw-moving-GT	2	22	0.04
hw-moving-T	2	22	0.04

results, contains images where the vehicle is stationary and the target is facing into the sun. In these images, the target is much closer to the camera, and a rectangular ROI around the target containing $\approx 63,840$ pixels is located in the lower right quadrant of the frame. The target again moves very little throughout the data set. One hundred sequential frames were selected from each data set, and then passed through the algorithm configurations for 5 iterations. The first frame of the data set was used to obtain the ROI for cropping and the time for this frame is not included in the results. The subsequent 99 frames were then used to obtain timing information. These results only capture the "steady-state" performance of the algorithm, that is, when the 6 markers are detected in each frame. The final FPS was computed by averaging the times for the 495 total frames.

It is evident from the data that all configurations out-perform the reference Matlab implementation by a significant amount. In order to obtain a better comparison, we updated the reference implementation with the optimizations from the OpenCV Version 1. Now, using the same *moving* data set, the Matlab implementation achieves 56 FPS. Against this result, the fully optimized embedded design (*hw-moving-GMT*) achieves a speed-up of 1.14, and the optimized embedded software version (*sw-moving-C*) achieves a speed-up of 1.09. Even the configuration without hardware thresholding (*hw-moving-GM*) has a speed-up of 1.07 against the improved

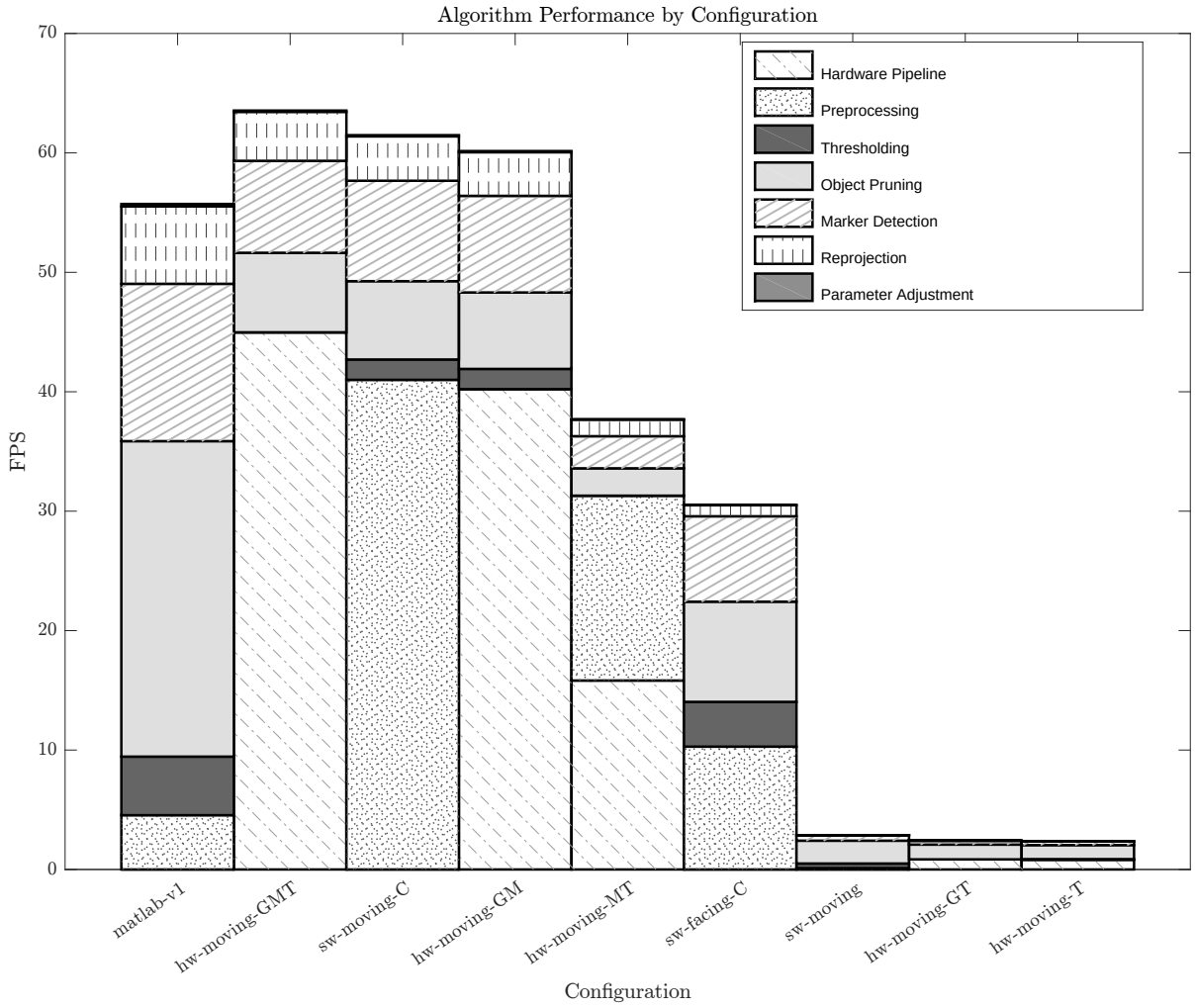


Figure 7.2: The average FPS for different combinations of hardware and software.

Matlab implementation. The 3 FPS performance difference between *hw-moving-GMT* and *sw-moving-C* arises from performing grayscale conversion and thresholding in software, and a slight increase in the computations for marker detection. This could be due to variations in the results of thresholding between software and hardware implementations. The 1 FPS difference between *sw-moving-C* and *hw-moving-GM* is due to the overhead incurred when reading the processed frame from the hardware pipeline back into the software application.

The configurations which perform worse than the improved Matlab algorithm may do so for a variety of reasons. The *sw-facing-C* configuration is identical to *sw-moving-C* except for the data set used, which indicates how much system performance depends on the size of the

ROI. A large ROI means more data to process, which consumes more time in software. The *hw-moving-MT* configuration has the overhead of receiving pixels from the hardware, as well as adding grayscale conversion in software, and as Figure 7.2 shows, the proportion of the total time spent converting the image to grayscale is close to the amount of time required for the hardware pipeline.

The last class of configurations which perform poorly are those without ROI cropping. The *sw-moving* configuration must convert the entire image to grayscale, threshold the entire image, and then locate and remove contours from the entire image. However, this configuration still performs better than the *hw-moving-GT* and *hw-moving-T* configurations, which indicates moving data from the destination frame buffer back into the software application incurs significant overhead.

As these results show, the performance of our algorithm degrades as more data must be copied from the output frame buffer, so as the distance from the target to the camera decreases, the amount of processing time increases. Additionally, when the algorithm is unable to locate six centers in a frame, the ROI is expanded to process the whole image until six frames are again located. When this happens, the performance of the system will be at about that of the *hw-moving-GT* configuration, which is about 2 FPS.

Table 7.4: Detailed configuration information.

Configuration	Data set[1] Name	Images	Grayscale	Masking	Cropping	Adaptive Thresholding
matlab-v1	Moving	4000-4099	sw	–	sw	sw
hw-moving-GMT	Moving	4000-4099	hw	hw	hw[2]	hw
sw-moving-C	Moving	4000-4099	sw	–	sw	sw
hw-moving-GM	Moving	4000-4099	hw	hw	hw	sw
hw-moving-MT	Moving	4000-4099	sw	hw	hw	hw
sw-facing-C	Facing Sun - combined	700-799	sw	–	sw	sw
sw-moving	Moving	4000-4099	sw	–	–	sw
hw-moving-GT	Moving	4000-4099	hw	–	hw[3]	hw
hw-moving-T	Moving	4000-4099	sw	–	hw[3]	hw

[1] All images are from the distortion-corrected data sets.
[2] Hardware "cropping" simply reads a subset of the image out of the destination frame buffer, rather than the whole image.
[3] In this case, there is effectively no cropping, since the whole image is read out of the destination frame buffer.

CHAPTER 8. CONCLUSION AND FUTURE WORK

Ellipse detection algorithms are frequently too computationally complex to process frames at the camera frame rate on typical processors in embedded computer vision systems. However, it is possible to overcome the computational complexities of an ellipse detection algorithm by using both CPU and FPGA architectures on an SoC, as demonstrated by the embedded ellipse detection system presented in this thesis. A Xilinx Zynq SoC is used to accelerate the computationally-expensive portions of the algorithm in the FPGA logic, utilizing a pixel-streaming architecture to apply grayscale conversion, ROI masking, and thresholding. Once the processed frame reaches the application software running on the ARM CPU, contours are extracted and ellipses are detected and fit using a least squares method from the OpenCV library. Our design achieves a frame rate of 64 FPS when at least six ellipses are detected in a frame, which is a speed-up of 1.14× over an optimized Matlab implementation. The system was first tested on an Avnet ZedBoard and then deployed on a Zynq-based camera system. Our system was successfully utilized in a controlled road construction environment, and, to the best of our knowledge, this thesis is the first work demonstrating an embedded ellipse detection system capable of processing HD resolution (1920×1080) images at the camera frame rate.

It was shown our system suffers a major blow to performance when fewer than six ellipses are located in the current frame. Our plans for future work include accounting for this weakness by investigating methods to utilize information from previous frame(s) to approximate locations of missing ellipses without sacrificing the frame rate. Deeper investigation into the memory transfer bottleneck from the PL to the PS could also result in performance benefits. Lastly, it would be useful to explore alternative ellipse detection algorithms and then compare the performance of the different methods, both in terms of execution time and correctness of the fitted ellipses. For such an analysis, a better ground truth data set would be beneficial.

REFERENCES

[1] R. Y. D. Xu and M. Kemp, "Fitting multiple connected ellipses to an image silhouette hierarchically," *IEEE Trans. Image Process.*, vol. 19, no. 7, pp. 1673–1682, Jul. 2010.

[2] M. P. Segundo, L. Silva, O. R. P. Bellon, and C. C. Queirolo, "Automatic face segmentation and facial landmark detection in range images," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 40, no. 5, pp. 1319–1330, Oct. 2010.

[3] C. T. Chou, S. W. Shih, W. S. Chen, V. W. Cheng, and D. Y. Chen, "Non-orthogonal view iris recognition system," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 20, no. 3, pp. 417–430, Mar. 2010.

[4] C. R. Del-Blanco, F. Jaureguizar, and N. García, "An efficient multiple object detection and tracking framework for automatic counting and video surveillance applications," *IEEE Trans. Consum. Electron.*, vol. 58, no. 3, pp. 857–862, Aug. 2012.

[5] G. Zhang, D. S. Jayas, and N. D. White, "Separation of touching grain kernels in an image by ellipse fitting algorithm," *Biosystems Engineering*, vol. 92, no. 2, pp. 135–142, 2005.

[6] C. Benedek, O. Krammer, M. Janóczki, and L. Jakab, "Solder paste scooping detection by multilevel visual inspection of printed circuit boards," *IEEE Trans. Ind. Electron.*, vol. 60, no. 6, pp. 2318–2331, Jun. 2013.

[7] N. Kharma *et al.*, "Automatic segmentation of cells from microscopic imagery using ellipse detection," *IET Image Processing*, vol. 1, no. 1, 39–47(8), Mar. 2007.

[8] H. Su, F. Xing, J. D. Lee, C. A. Peterson, and L. Yang, "Automatic myonuclear detection in isolated single muscle fibers using robust ellipse fitting and sparse representation," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 11, no. 4, pp. 714–726, Jul. 2014.

[9] T. Behrens, K. Rohr, and H. S. Stiehl, "Robust segmentation of tubular structures in 3-D medical images by parametric object detection and tracking," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 33, no. 4, pp. 554–561, Aug. 2003.

[10] S. Malassiotis and M. G. Strintzis, "Tracking the left ventricle in echocardiographic images by learning heart dynamics," *IEEE Trans. Med. Imag.*, vol. 18, no. 3, pp. 282–290, Mar. 1999.

[11] J. Feulner *et al.*, "A probabilistic model for automatic segmentation of the esophagus in 3-D CT scans," *IEEE Trans. Med. Imag.*, vol. 30, no. 6, pp. 1252–1264, Jun. 2011.

[12] A. M. Fernandes, "Study on the automatic recognition of oceanic eddies in satellite images by ellipse center detection – the Iberian Coast case," *IEEE Trans. Geosci. Remote Sens.*, vol. 47, no. 8, pp. 2478–2491, Aug. 2009.

[13] B. Xiong, J. M. Chen, G. Kuang, and N. Kadowaki, "Estimation of the repeat-pass ALOS PALSAR interferometric baseline through direct least-square ellipse fitting," *IEEE Trans. Geosci. Remote Sens.*, vol. 50, no. 9, pp. 3610–3617, Sep. 2012.

[14] S. Xiaofang and G. Wencheng, "A sun spot center orientation method based on ellipse fitting in the application of CPV solar tracker," in *Int. Conf. Intelligent System Design and Engineering Application*, vol. 1, Oct. 2010, pp. 175–178.

[15] M. Zakrzewski *et al.*, "Quadrature imbalance compensation with ellipse-fitting methods for microwave radar physiological sensing," *IEEE Trans. Microw. Theory Techn.*, vol. 62, no. 6, pp. 1400–1408, Jun. 2014.

[16] C. R. Rojas, P. Zetterberg, and P. Händel, "Transceiver inphase/quadrature imbalance, ellipse fitting, and the universal software radio peripheral," *IEEE Trans. Instrum. Meas.*, vol. 60, no. 11, pp. 3629–3639, Nov. 2011.

[17] D. Liu and J. Liang, "A Bayesian approach to diameter estimation in the diameter control system of silicon single crystal growth," *IEEE Trans. Instrum. Meas.*, vol. 60, no. 4, pp. 1307–1315, Apr. 2011.

[18] P. M. Ramos, F. M. Janeiro, M. Tlemçani, and A. C. Serra, "Recent developments on impedance measurements with DSP-based ellipse-fitting algorithms," *IEEE Trans. Instrum. Meas.*, vol. 58, no. 5, pp. 1680–1689, May 2009.

[19] A. Y. S. Chia, S. Rahardja, D. Rajan, and M. K. Leung, "A split and merge based ellipse detector with self-correcting capability," *IEEE Trans. Image Process.*, vol. 20, no. 7, pp. 1991–2006, Jul. 2011.

[20] M. Russell and S. Fischaber, "OpenCV based road sign recognition on Zynq," in *IEEE 11th Int. Conf. Industrial Informatics*, Jul. 2013, pp. 596–601.

[21] T. Kryjak, M. Komorkiewicz, and M. Gorgoń, "Real-time hardware–software embedded vision system for ITS smart camera implemented in Zynq SoC," *J. Real-Time Image Processing*, pp. 1–37, 2016.

[22] S. Zhao *et al.*, "A robust real-time vision system for autonomous cargo transfer by an unmanned helicopter," *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1210–1219, Feb. 2015.

[23] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Commun. ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972.

[24] D. Ballard, "Generalizing the Hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, no. 2, pp. 111–122, 1981.

[25] P. Nair and A. Saunders, "Hough transform based ellipse detection algorithm," *Pattern Recognition Letters*, vol. 17, no. 7, pp. 777–784, 1996.

[26] N. Guil and E. Zapata, "Lower order circle and ellipse Hough transform," *Pattern Recognition*, vol. 30, no. 10, pp. 1729–1744, 1997.

[27] R. A. McLaughlin, "Randomized Hough transform: improved ellipse detection with comparison," *Pattern Recognition Letters*, vol. 19, no. 3, pp. 299–305, 1998.

[28] C.-F. Chien, Y.-C. Cheng, and T.-T. Lin, "Robust ellipse detection based on hierarchical image pyramid and Hough transform," *J. Opt. Soc. Am. A*, vol. 28, no. 4, pp. 581–589, Apr. 2011.

[29] R. C. Bolles and M. A. Fischler, "A RANSAC-based approach to model fitting and its application to finding cylinders in range data.," in *IJCAI*, vol. 1981, 1981, pp. 637–643.

[30] G. Song and H. Wang, "A fast and robust ellipse detection algorithm based on pseudo-random sample consensus," in *Proc. 12th Int. Conf. Computer Analysis of Images and Patterns*, W. G. Kropatsch, M. Kampel, and A. Hanbury, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 669–676.

[31] F. Mai, Y. Hung, H. Zhong, and W. Sze, "A hierarchical approach for fast and robust ellipse extraction," *Pattern Recognition*, vol. 41, no. 8, pp. 2512–2524, 2008.

[32] R. N. Dave, "Generalized fuzzy c-shells clustering and detection of circular and elliptical boundaries," *Pattern Recognition*, vol. 25, no. 7, pp. 713–721, 1992.

[33] I. Gath and D. Hoory, "Fuzzy clustering of elliptic ring-shaped clusters," *Pattern Recognition Letters*, vol. 16, no. 7, pp. 727–741, 1995.

[34] H. Frigui and R. Krishnapuram, "A comparison of fuzzy shell-clustering methods for the detection of ellipses," *IEEE Trans. Fuzzy Syst.*, vol. 4, no. 2, pp. 193–199, May 1996.

[35] J. Porrill, "Fitting ellipses and predicting confidence envelopes using a bias corrected Kalman filter," *Image and Vision Computing*, vol. 8, no. 1, pp. 37–41, 1990.

[36] A. W. Fitzgibbon, M. Pilu, and R. B. Fisher, "Direct least squares fitting of ellipses," in *Proc. 13th Int. Conf. Pattern Recognition*, vol. 1, Aug. 1996, 253–257 vol.1.

[37] P. L. Rosin, "A note on the least squares fitting of ellipses," *Pattern Recognition Letters*, vol. 14, no. 10, pp. 799–808, 1993.

[38] ——, "Analysing error of fit functions for ellipses," *Pattern Recognition Letters*, vol. 17, no. 14, pp. 1461–1470, 1996.

[39] W. Gander, G. H. Golub, and R. Strebel, "Least-squares fitting of circles and ellipses," *BIT Numerical Mathematics*, vol. 34, no. 4, pp. 558–578, 1994.

[40] R. Halíř and J. Flusser, "Numerically stable direct least squares fitting of ellipses," in *Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualization*, Citeseer, vol. 98, 1998, pp. 125–132.

[41] E. S. Maini, "Enhanced direct least square fitting of ellipses," *Int. J. Pattern Recognition and Artificial Intelligence*, vol. 20, no. 06, pp. 939–953, 2006.

[42] K. Kanatani and P. Rangarajan, "Hyper least squares fitting of circles and ellipses," *Computational Statistics & Data Analysis*, vol. 55, no. 6, pp. 2197–2208, 2011.

[43] S. J. Ahn, W. Rauh, and M. Recknagel, "Ellipse fitting and parameter assessment of circular object targets for robot vision," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, vol. 1, 1999, 525–530 vol.1.

[44] D. S. Barwick, "Very fast best-fit circular and elliptical boundaries by chord data," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 6, pp. 1147–1152, Jun. 2009.

[45] P.-Y. Yin, "A new circle/ellipse detector using genetic algorithms," vol. 20, no. 7, pp. 731–740, 1999.

[46] K. Kanatani and Y. Sugaya, "Compact algorithm for strictly ML ellipse fitting," in *19th Int. Conf. Pattern Recognition*, Dec. 2008, pp. 1–4.

[47] J. Illingworth and J. Kittler, "A survey of the Hough transform," *Computer Vision, Graphics, and Image Processing*, vol. 44, no. 1, pp. 87–116, 1988.

[48] V. Leavers, "Which Hough transform?" *CVGIP: Image Understanding*, vol. 58, no. 2, pp. 250–264, Sep. 1993.

[49] P. Mukhopadhyay and B. B. Chaudhuri, "A survey of Hough transform," *Pattern Recognition*, vol. 48, no. 3, pp. 993–1010, 2015.

[50] L. Xu, E. Oja, and P. Kultanen, "A new curve detection method: randomized Hough transform (RHT)," *Pattern Recognition Letters*, vol. 11, no. 5, pp. 331–338, 1990.

[51] H. K. Yuen, J. Illingworth, and J. Kittler, "Detecting partially occluded ellipses using the Hough transform," *Image and Vision Computing*, vol. 7, no. 1, pp. 31–37, 1989.

[52] H. Li, M. A. Lavin, and R. J. Le Master, "Fast Hough transform: a hierarchical approach," *Computer Vision, Graphics, and Image Processing*, vol. 36, no. 2, pp. 139–161, 1986.

[53] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981.

[54] W. Cai, Q. Yu, and H. Wang, "A fast contour-based approach to circle and ellipse detection," in *5th World Congr. Intelligent Control and Automation*, vol. 5, Jun. 2004, 4686–4690 Vol.5.

[55] Y. Sugaya, "Ellipse detection by combining division and model selection based integration of edge points," in *4th Pacific-Rim Symp. Image and Video Technology*, Nov. 2010, pp. 64–69.

[56] K. Kanatani, Y. Sugaya, and Y. Kanazawa, *Ellipse Fitting for Computer Vision: Implementation and Applications*, ser. Synthesis Lectures on Computer Vision. Morgan & Claypool Publishers, 2016.

[57] F. L. Bookstein, "Fitting conic sections to scattered data," *Computer Graphics and Image Processing*, vol. 9, no. 1, pp. 56–71, Jan. 1979.

[58] A. W. Fitzgibbon and R. B. Fisher, "A buyer's guide to conic fitting," *DAI Research paper*, 1996.

[59] G. Taubin, "Estimation of planar curves, surfaces, and nonplanar space curves defined by implicit equations with applications to edge and range image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 11, pp. 1115–1138, Nov. 1991.

[60] D. K. Prasad, M. K. Leung, and C. Quek, "ElliFit: an unconstrained, non-iterative, least squares based geometric ellipse fitting method," *Pattern Recognition*, vol. 46, no. 5, pp. 1449–1465, 2013.

[61] Y. Ito, K. Ogawa, and K. Nakano, "Fast ellipse detection algorithm using Hough transform on the GPU," in *2nd Int. Conf. Networking and Computing*, Nov. 2011, pp. 313–319.

[62] J. K. Lee, B. A. Wood, and T. S. Newman, "Very fast ellipse detection using GPU-based RHT," in *19th Int. Conf. Pattern Recognition*, Dec. 2008, pp. 1–4.

[63] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, E. L. Zapata, and N. Guil, "Load balancing versus occupancy maximization on graphics processing units: the generalized Hough transform as a case study," *Int. J. High Performance Computing Applications*, vol. 25, no. 2, pp. 205–222, 2011.

[64] S. Martelli, R. Marzotto, A. Colombari, and V. Murino, "FPGA-based robust ellipse estimation for circular road sign detection," in *IEEE Comput. Soc. Conf. Computer Vision and Pattern Recognition - Workshops*, Jun. 2010, pp. 53–60.

[65] K. Dohi, Y. Hatanaka, K. Negi, Y. Shibata, and K. Oguri, "Deep-pipelined FPGA implementation of ellipse estimation for eye tracking," in *22nd Int. Conf. Field Programmable Logic and Applications*, Aug. 2012, pp. 458–463.

[66] M. G. Albanesi, M. Ferretti, and D. Rizzo, "Benchmarking Hough transform architectures for real-time," *Real-Time Imaging*, vol. 6, no. 2, pp. 155–172, 2000.

[67] D. G. Bailey, "Considerations for hardware Hough transforms," *Image and Vision Computing, New Zealand, Australia*, 2011.

[68] S. M. Karabernou and F. Terranti, "Real-time FPGA implementation of Hough transform using gradient and CORDIC algorithm," *Image and Vision Computing*, vol. 23, no. 11, pp. 1009–1017, 2005.

[69] P. Lee and A. Evagelos, "An implementation of a multiplierless Hough transform on an FPGA platform using hybrid-log arithmetic," in *Proc. SPIE*, vol. 6811, 2008, 68110G–68110G–10.

[70] H. Koshimizu and M. Numada, "FIHT2 algorithm: a fast incremental Hough transform," *IEICE Trans. Information and Systems*, vol. 74, no. 10, pp. 3389–3393, 1991.

[71] S. Tagzout, K. Achour, and O. Djekoune, "Hough transform algorithm for FPGA implementation," *Signal Processing*, vol. 81, no. 6, pp. 1295–1301, 2001.

[72] T. Maruyama, "Real-time computation of the generalized Hough transform," in *Proc. 14th Int. Conf. Field Programmable Logic and Application*, J. Becker, M. Platzner, and S. Vernalde, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 980–985.

[73] G. J. Tu, M. K. Hansen, P. Kryger, and P. Ahrendt, "Automatic behaviour analysis system for honeybees using computer vision," *Computers and Electronics in Agriculture*, vol. 122, pp. 10–18, 2016.

[74] M. Fornaciari, A. Prati, and R. Cucchiara, "A fast and effective ellipse detector for embedded vision applications," *Pattern Recognition*, vol. 47, no. 11, pp. 3693–3708, 2014.

[75] J. Faigl, T. Krajník, J. Chudoba, L. Přeučil, and M. Saska, "Low-cost embedded system for relative localization in robotic swarms," in *IEEE Int. Conf. Robotics and Automation*, May 2013, pp. 993–998.

[76] T. Krajník, M. Nitsche, J. Faigl, T. Duckett, M. Mejail, and L. Přeučil, "External localization system for mobile robotics," in *16th Int. Conf. Advanced Robotics*, Nov. 2013, pp. 1–6.

[77] D. C. Brown, "Close-range camera calibration," *Photogrammetric Engineering*, vol. 37, no. 8, pp. 855–866, 1971.

[78] ——, "Decentering distortion of lenses," *Photometric Engineering*, vol. 32, no. 3, pp. 444–462, 1966.

[79] A. Kaehler and G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2016.

[80] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern Recognition*, vol. 33, no. 2, pp. 225–236, 2000.

[81] F. Shafait, D. Keysers, and T. M. Breuel, "Efficient implementation of local adaptive thresholding techniques using integral images," in *Proc. SPIE*, vol. 6815, 2008, pp. 681510–681510–6.

[82] S. Suzuki and K. Abe, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 3, no. 1, pp. 32–46, 1985.

[83] Itseez, *OpenCV*, 2017. [Online]. Available: `opencv.org` (visited on 02/07/2017).

[84] Xilinx, San Jose, CA, USA, *UG585: Zynq-7000 all programmable SoC technical reference manual*, (2015). [Online]. Available: `www.xilinx.com`.

[85] DENX Software Engineering, *Das U-Boot: the universal boot loader*, 2016. [Online]. Available: `www.denx.de/wiki/U-Boot` (visited on 02/07/2017).

[86] Xilinx, San Jose, CA, USA, *PG020: AXI video direct memory access v6.2*, (2015). [Online]. Available: `www.xilinx.com`.

[87] *AMBA 4 AXI4-Stream Protocol*, ARM IHI 0051A, 2010. [Online]. Available: `www.arm.com`.

[88] Xilinx, San Jose, CA, USA, *UG1037: Vivado Design Suite AXI reference guide*, (2015). [Online]. Available: `www.xilinx.com`.