IMPLEMENTATION OF AN AFFINE-INVARIANT FEATURE DETECTOR IN FIELD-PROGRAMMABLE GATE ARRAYS

BY

CRISTINA CABANI

AUGUST 2006

A THESIS SUBMITTED IN CONFORMITY WITH THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE
GRADUATE DEPARTMENT OF THE EDWARD S. ROGERS SR.
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF TORONTO

To my mother

Abstract

Implementation of an Affine-Invariant Feature Detector in Field-Programmable Gate Arrays

Cristina Cabani

Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

2006

Feature detectors are algorithms that can locate and describe points or regions of 'interest' — or *features*— in an image. As the complexity of the feature detection algorithm increases, so does the amount of time and computer resources required to perform it. For this reason, feature detectors are increasingly being ported to hardware circuits such as field-programmable gate arrays (FPGAs), where the inherent parallelism of these algorithms can be exploited to provide significant increase in speed.

This work describes an FPGA-based implementation of the Harris-Affine feature detector introduced by Mikolajczyk and Schmid [36, 37]. The system is implemented on the Transmogrifier-4, a prototyping platform developed at the University of Toronto that includes four Altera Stratix S80 FPGAs and NTSC/VGA video interfaces. The system achieves a speed of 90–9000 times the speed of an equivalent software implementation, allowing it to process standard video (640×480 pixels) at 30 frames per second.

Acknowledgements

I would like to express my gratitude to W. James MacLean for his guidance and support throughout this project. He has been a patient and enthusiastic supervisor and has provided a wealth of new and exciting ideas. Many thanks to my friends in the Vision and Image Dynamics Lab for making my time at U of T so much more enjoyable. In particular, I would like to thank Divyang Masrani, Leyla Imanirad and Siraj Sabihuddin for all their encouragement and their helpful comments on the hardware design and implementation.

I am grateful to Joshua Fender and Dave Galloway for providing design examples and technical support for the Transmogrifier-4. Special thanks to Eugenia Distefano for keeping our computer systems running smoothly. I would also like to thank Communications and Information Technology Ontario (CITO) and the Department of Computer and Electrical Engineering at the University of Toronto for providing the financial support that made this project possible.

Finally, I am deeply grateful to Michael Daly, whose unconditional support and encouragement have been a driving force during my graduate studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Human vision is a complex combination of physical, psychological and neurological processes that allows us to interact with our environment. We use vision effortlessly to detect, identify and track objects, to navigate and to create a conceptual map of our surroundings.

The goal of computer vision is to design computer systems that are capable of performing these tasks both accurately and efficiently using a minimal amount of time and resources. Current computer vision systems are far from matching the flexibility of biological visual systems, however they have been successfully applied in areas such as manufacturing, medicine and robotics to perform object recognition, scene reconstruction, motion tracking, actuator control, image retrieval and data mining, among many others.

Part of the research in computer vision focuses on developing algorithms that can locate and describe points or regions of 'interest'— or *features*— in an image. The idea is that if a structure in an image (such as part of an object or a texture) can be described by a limited set of features, and this set is robust to changes in viewing conditions, then the features can be used to identify and match the same structure in different images.

Once a set of features is detected in an image, the characteristics of the image

around the features can be encoded into description vectors. These description vectors, for example, can be computed independently for two images and then used to determine if the same structures are present in the two images. Description vectors can also be used to index into a database of object descriptors to perform object recognition. Features and their associated descriptors are often computed in the preliminary stages of systems for object recognition, object tracking, motion analysis and video indexing, among others.

What constitutes a feature depends to a large extent on what is the intended use for the information that is extracted from the images. For example, different applications may have different requirements for the robustness of the features to changes in viewing conditions. In general, feature detectors try to achieve invariance to changes in illumination, scale, location, 2D rotation and even affine or perspective transformations. In addition, features should be distinctive so that features corresponding to different structures can be easily distinguished.

As the complexity of the vision task increases, so does the amount of time and computer resources required to perform it. In particular, embedded vision systems, like the ones used in autonomous robot navigation and inspection of manufacturing processes, need to process large amounts of data in real-time (1–200 frames per second depending on the task). For this reason, many vision algorithms have been ported to integrated circuits such as Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). Integrated circuits, often referred to simply as *hardware*, can increase the speed of vision algorithms by one to three orders of magnitude when compared to the speed achieved in general-purpose processors.

Hardware implementations of vision algorithms can achieve this increase in speed because of the inherent parallelism of many of the operations involved in processing an image. For example, convolution of an image with a two-dimensional filter is implemented in a general-purpose processor as a series of multiplications and additions where only one coefficient of the filter is processed at a time. Thus, several cycles

are required to produce a result for a single image pixel. On the other hand, in a hardware implementation all coefficients in the filter could potentially be processed at the same time, as long as there are enough resources to implement the necessary multipliers and adders. Moreover, since many of the processes in a vision system are relatively independent of each other, the system can be pipelined so that several processes are executed at the same time, which greatly increases the throughput of data.

The amount of resources available to implement operations in hardware is often limited by factors such as cost, weight and power consumption. Thus, it is important to design an implementation of the algorithm that is carefully crafted to balance the need for accuracy and numerical precision with an efficient use of the hardware resources. For this purpose, many hardware designs use a fixed-point representation instead of the floating-point representation used in general-purpose processors.

Historically, most commercial computer vision systems have been implemented in ASICs. ASICs can achieve higher speeds and less power consumption than FPGAs because all logic and routing in the circuit is fabricated specially for the specific application. For this same reason, however, it takes significant time and effort to develop an ASIC-based system and no changes can be made to the design once the circuit has been fabricated. FPGAs, on the other hand, are becoming increasing popular for computer vision applications because they can be reprogrammed. This can shorten the development time of a system considerably and allow prototype systems to be deployed and tested under real conditions. In many cases, corrections and additional features can be implemented in hours or days.

The extensive use of feature detectors as preliminary stages of vision applications, coupled with the capacity and speed of current FPGAs, have led to the development of smart camera systems with integrated feature detection stages. These systems can preprocess incoming video from a camera and provide real-time information to subsequent processing stages.

This thesis describes an FPGA-based implementation of the affine invariant feature detector introduced by Mikolajczyk and Schmid [36, 37], referred to in the literature as the Harris-Affine detector. The thesis is organized as follows: Chapter 2 introduces the feature detector, field-programmable gate arrays and the Transmogrifier-4, and gives a brief review of the design methodology. Chapter 3 provides a detailed description of the design of the detector, including a discussion on the fixed-point approximations. Chapter 4 presents the results obtained from the completed system and Chapter 5 presents the conclusions of the project and discusses possible directions for future work.

Chapter 2

Background

This chapter introduces some of the concepts applied throughout this thesis. Section 2.1 provides a brief overview of the methods developed for feature detection encountered in the literature, with focus on those based on image intensity that achieve invariance to scale and affine transformations. Section 2.2 describes the Harris-Affine feature detector. Section 2.3 introduces field-programmable gate arrays and the Transmogrifier-4. Section 2.4 discusses some of the existing FPGA-based vision systems, in particular those that implement feature detection in real time. Finally, Section 2.5 presents the design methodology.

2.1 Scale and affine invariant feature detectors

The literature on feature detectors is extensive. The detection algorithms can be based on image characteristics such as contours, intensity and phase[1], as well as on parametric models. Despite their differences, all methods attempt to achieve invariance to changes in viewpoint to facilitate the description and matching of objects across images.

Most intensity-based feature detectors can be traced back to the work of Moravec

[1]The term 'phase' refers to the phase response of a complex-value filter applied to an image.

[40]. His method observes the average changes in image intensity, over a local window, that result from shifting the window over the image. The features are the points at which the image intensity changes in more than one direction (corner-like structures). Harris and Stephens [21] improved Moravec's idea to make the detection more robust to noise and nearby edges. Their method uses the second moment matrix $M(x, y)$ to describe the image gradient in a Gaussian window $w(x, y)$ centred at (x, y):

$$M(x,y) = \begin{bmatrix} w(x,y) * (I_x(x,y))^2 & w(x,y) * (I_x(x,y)I_y(x,y)) \\ w(x,y) * (I_x(x,y)I_y(x,y)) & w(x,y) * (I_y(x,y))^2 \end{bmatrix} \tag{2.1}$$

where I_x and I_y are the derivatives of the image in the x- and y-directions. The features are selected at points where the second moment matrix has two large eigenvalues, indicating significant intensity changes in more than a single direction. The detector is robust to rotation and translation, however its performance progressively degrades as the change in scale becomes more significant because the image derivatives are sensitive to changes in the size of the local structure.

In general, the size of objects can vary significantly between images and thus algorithms that search for features at a single scale miss important information present at other scales. Many of the methods that achieve invariance to scale changes analyze images at multiple scales using *image pyramids* [1]. Image pyramids are hierarchical structures that represent an image at different resolutions. The different 'levels' of the pyramid are formed by convolving the original image with filters at different scales. Figure 2.1 shows a *Gaussian pyramid*, in which each level is formed by convolving the previous level with a Gaussian filter at scale $\sigma = 2$ and then subsampling. Since Gaussian filters attenuate high-frequency components, each level can be subsampled to reduce the amount of effort required to compute the next level. The number of levels and the type of filters used depend on the application. Often different levels of the pyramid are combined to form different representations, as in the case of the *difference-of-Gaussians pyramid*, where each level is formed by subtracting adjacent levels of a Gaussian pyramid.

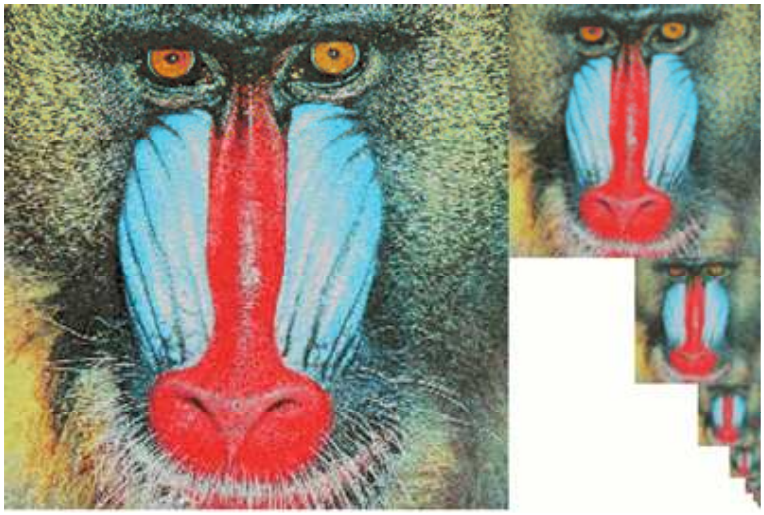


Figure 2.1: Gaussian pyramid. The first level of the pyramid (left) is the original image. Subsequent levels are formed by convolving the previous level with a Gaussian filter of standard deviation $\sigma = 2$. Image courtesy of [56]

There are numerous approaches that explore ways to detect features that are invariant to scale changes. Crowley and Parker [10] presented a multiscale representation of shapes in an image, constructed by detecting peaks and ridges in a pyramid of difference-of-Gaussians. Their algorithm forms a multiscale tree that describes shape by linking adjacent peaks and ridges across different levels of the pyramid. Lindeberg has done extensive work on the optimal scales for feature selection [7, 27, 26, 28]. He proposed a method to detect blob-like structures over scale space [26] and later studied how feature detection with automatic scale selection can be formulated for various kinds of features based on scale spaces formed by normalized differential operators [28]. Shokoufandeh, Marsic and Dickinson [48] developed a scale-invariant method that captures the salient regions of an object using a wavelet transform at different scales. Lowe [30, 31] selects image locations that are maxima, across scales, in a difference-of-Gaussians pyramid. The location of the features is refined by fitting a 3D quadratic function to their location and scale and then enforcing the condition that the Hessian of the image intensity present two large eigenvalues. The features are also assigned an orientation based on the the dominant directions of the local

gradients. Mikolajczyk and Schmid [35] extended Harris's approach to achieve invariance to scale. Their method uses a scale-normalized second moment matrix and selects points at which the matrix presents two large eigenvalues. The characteristic scale for each point is the scale at which the Laplacian achieves an extremum.

Tuytelaars and Van Gool developed two different affine-invariant detectors in the context of image retrieval and wide-baseline stereo matching. The first method [53] fits a parallelogram to a region bounded by a corner and two nearby edges. Photometric measures of the texture covered by the parallelogram determine its final shape. The second method [54] looks for points that are extrema of an intensity function and that are located along the rays that originate from a local maxima in intensity. The algorithm then fits an ellipse to the shape formed by these points. In these methods, the regions covered by the parallelograms and ellipses are invariant to affine transformations. Matas *et al.* [34] proposed a method that finds the regions in an image with intensities above a threshold, for a range of thresholds. The areas found at different thresholds are combined into connected components (maximally stable extremal regions — MSER) using the watershed algorithm. Schaffalitzky and Zisserman [43] extended Mikolajczyk's Harris-Laplace corner detector [35] to iteratively estimate the affine shape of the region around the features. Mikolajczyk [36] built over the latter method to include iterative refinement of the scale and location of the features. This method is the Harris-Affine detector presented next.

2.2 The Harris-Affine feature detector

The system described in this thesis implements the Harris-Affine interest point detector introduced by Mikolajczyk and Schmid in [36] and [37]. The Harris-Affine detector combines the scale-invariant Harris-Laplace approach [35] with an iterative procedure for computing the affine shape of an image region in the neighbourhood of the features.

The Harris-Laplace detector computes a multiscale representation of the Harris corner detector [21] and selects image locations at which the local Laplacian (Equation 2.4) is maximum over scales. This combination provides a set of distinctive points that are robust to changes in the amount of illumination, scale, rotation and translation, as well as limited changes in viewpoint. The Harris-Laplace detector is extended to deal with affine transformations by iteratively refining the location, scale and shape of the feature until its neighbourhood satisfies an isotropy condition.

2.2.1 The Harris-Laplace detector

The Harris-Laplace detector uses the scale-normalized second moment matrix $\mu(\mathbf{x}, \sigma_I, \sigma_D)$ to describe the variance of the image intensities around the point $\mathbf{x}$:

$$\mu(\mathbf{x}, \sigma_I, \sigma_D) = \sigma_D^2 \, g(\sigma_I) * \begin{bmatrix} L_x^2(\mathbf{x}, \sigma_D) & L_x L_y(\mathbf{x}, \sigma_D) \\ L_x L_y(\mathbf{x}, \sigma_D) & L_y^2(\mathbf{x}, \sigma_D) \end{bmatrix}. \tag{2.2}$$

The differentiation scale σ_D and the integration scale σ_I describe, respectively, the amount of smoothing used to compute the derivatives and the size of the region over which the elements of the second moment matrix are accumulated. The quantities $L_x(\mathbf{x}, \sigma_D)$ and $L_y(\mathbf{x}, \sigma_D)$ are the first derivatives of the image at $\mathbf{x}$ in the x- and y-directions computed with a Gaussian derivative at scale σ_D.

The algorithm finds points at which the image intensity changes in more than one direction, or equivalently, points at which the second moment matrix has two large eigenvalues. The explicit computation of the eigenvalues is avoided by using the trace and determinant of $\mu(\mathbf{x}, \sigma_I, \sigma_D)$ [21]. The latter are combined into the *cornerness* function,

$$\textit{cornerness} = \det(\mu(\mathbf{x}, \sigma_I, \sigma_D)) - \alpha \, \mathrm{trace}^2(\mu(\mathbf{x}, \sigma_I, \sigma_D)), \tag{2.3}$$

where α is a positive parameter usually in the range $0 \leq \alpha \leq 0.25$.

The *cornerness* function is evaluated for every pixel in the image and over a range of scales. At each scale (σ_I, σ_D), a point is selected as a corner if its *cornerness* is a

maximum in its 8-point neighbourhood.

In general, an image structure is present over a range of scales. To avoid having representations of the same structure at various scale, the Harris-Laplace detector selects the scale at which the absolute value of the Laplacian $\mathrm{LoG}(\mathbf{x}, \sigma_I)$ (Equation 2.4) achieves a maximum over scales,

$$|\mathrm{LoG}(\mathbf{x}, \sigma_I)| = \sigma_I^2 \, |L_{xx}(\mathbf{x}, \sigma_I) + L_{yy}(\mathbf{x}, \sigma_I)|. \tag{2.4}$$

2.2.2 Affine shape adaptation

When a planar object is imaged from two different viewpoints, a point $\mathbf{x} = (x, y)$ in one of the images is related to a point $\mathbf{x}' = (x', y')$ in the second image by a projective transformation [23] H_P such that

$$\begin{aligned} \mathbf{x}' &= H_P \, \mathbf{x} \\ \begin{bmatrix} x'_z \\ y'_z \\ z \end{bmatrix} &= \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \end{aligned} \tag{2.5}$$

and

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \frac{1}{z} \begin{bmatrix} x'_z \\ y'_z \\ z \end{bmatrix}. \tag{2.6}$$

In a projective transformation, like the one shown in Figure 2.2(b), parallel world lines appear as converging lines and the relative sizes of the objects in the image depend on their original position with respect to the camera, e.g., objects that are further away from the camera seem smaller.

In general, objects in the world do not lie entirely in a single plane, however a sufficiently small surface can be approximated as a set of coplanar points [37]. Moreover, a projective transformation of a smooth surface can be locally approximated by

(a) (b) (c)

Figure 2.2: Distortions arising from projective and affine transformations. (a) Tiled floor. (b) Projective transformation. (c) Affine transformation. Images courtesy of [22].

an affine transformation H_A [37] such that

$$\mathbf{x}' = H_A\,\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \mathbf{x}. \tag{2.7}$$

The vector $\mathbf{t} = [t_x, t_y]^T$ represents the translation between corresponding points. The matrix $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ accounts for rotation, sheer and scaling, and can be thought of as a composition of a rotation and a non-isotropic scaling:

$$A = R(\theta)\,R(-\phi)\,D\,R(\phi) \qquad \text{where } D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}. \tag{2.8}$$

In this expression $R(\theta)$ and $R(\phi)$ are rotations by θ and ϕ respectively. The factors λ_1 and λ_2 determine the scaling in the rotated x and y directions (rotated by an angle ϕ). Figure 2.3 illustrates the effect of these parameters on the affine transformation of a square region. In an affine transformation (Figure 2.2(c)), parallel world lines remain parallel, however angles are not preserved and circles are mapped to ellipses.

In the context of feature detection, the non-uniform scaling by λ_1 and λ_2 means that the scale chosen with the isotropic Harris-Laplace detector may not reflect the real transformation between two images. Moreover, the location of the maxima in the

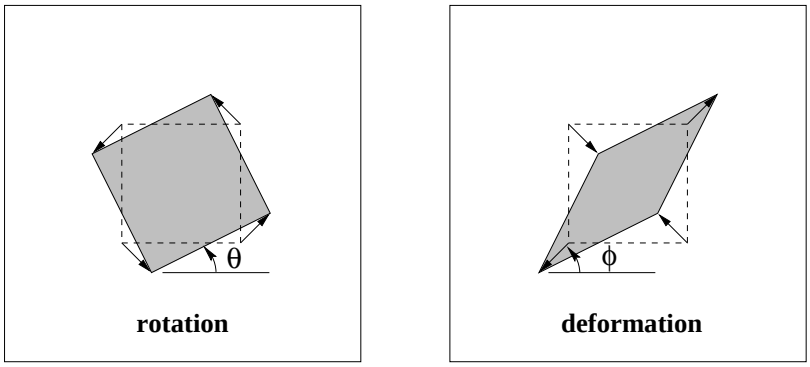


Figure 2.3: Effect of an affine transformation on a square region.
Images courtesy of [22].

Harris *cornerness* function varies depending on the scale, and as a result there is an additional error introduced in the location of the features. To avoid these problems, the features need to be computed in an affine scale-space formed by convolution of the image with non-isotropic, elliptical Gaussian kernels.

The Harris-Affine algorithm applies an affine-normalization concept introduced by Baumberg [3] and Lindeberg [29], in which the features are computed in a *transformed* image domain. Instead of applying non-isotropic Gaussian kernels to the original image, the image itself is warped based on the local second moment matrix (Equation 2.2), so that all subsequent location and scale computations are performed using circular Gaussian filters. This process is performed iteratively until the normalized regions around the feature points present isotropic intensity patterns. At each feature location, a 2×2 *shape adaptation matrix (U)*, derived from the local second moment matrix, describes the transformation that has to be applied to an image patch around the feature point to obtain the normalized region. The result of this process is that if corresponding points in two images are related by an affine transformation (or can be approximated as such), then the affine-normalized regions around these points are the same up to an arbitrary rotation matrix.

Figure 2.4 illustrates the result of normalizing image regions around corresponding feature points in two images. The images on the left show the regions, marked

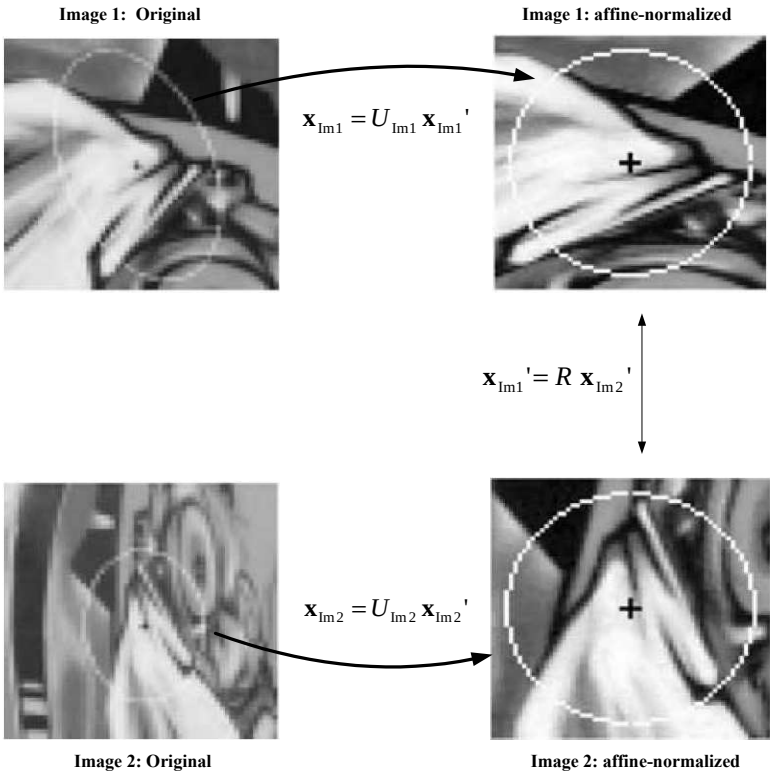


Figure 2.4: Affine-normalization based on the shape adaptation matrix. The images on the left show the corresponding regions in the original images (white ellipses), which are related by an affine transformation. The regions are affine-normalized by the transformations U_{Im1} and U_{Im2} so that the normalized regions are related by an arbitrary rotation matrix. Image modified from [37].

by white ellipses, in the original images, which are related to each other by an affine transformation. The regions are affine-normalized by transforming the image coordinates $\mathbf{x}_{Im1}$ and $\mathbf{x}_{Im2}$ with the matrices U_{Im1} and U_{Im2} respectively. The resulting normalized regions are related to each other by a rotation matrix.

2.2.3 Algorithm

The Harris-Affine algorithm is initialized with points extracted with the multiscale Harris detector. For each preliminary feature at location $\mathbf{x}^{(0)}$ and scale ${\sigma_I}^{(0)}$ the algorithm applies the following procedure:

1. Initialize the *shape adaptation matrix* U to the identity matrix: $U^{(0)} = I$.

2. Normalize an image region $W(\mathbf{x}_w)$ centred at $\mathbf{x}_w^{(k-1)} = U^{(k-1)^{-1}}\mathbf{x}^{(k-1)}$, where the superscript (k) denotes the k^{th} iteration.

3. Select the *integration scale* ${\sigma_I}^{(k)}$ at the point $\mathbf{x}^{(k-1)}$ that maximizes the absolute value of the Laplacian (Equation 2.4).

4. Select the *differentiation scale* ${\sigma_D}^{(k)} = s\,{\sigma_I}^{(k)}$ that maximizes the isotropy ratio $Q = \frac{\lambda_{min}(\mu)}{\lambda_{max}(\mu)}$, where $s \in [0.5, \ldots, 0.75]$ and $\mu = \mu(\mathbf{x}_w^{(k-1)}, {\sigma_I}^{(k)}, {\sigma_D}^{(k)})$ is the second moment matrix at the point $\mathbf{x}_w^{(k-1)}$ computed from the image region W.

5. Detect the *spatial localization* $\mathbf{x}_w^{(k)}$ of a maximum of the Harris *cornerness* function (Equation 2.3) around $\mathbf{x}_w^{(k-1)}$ and compute the location of the feature in the original image domain $\mathbf{x}^{(k)}$:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + U^{(k-1)}\,(\mathbf{x}_w^{(k)} - \mathbf{x}_w^{(k-1)}) \tag{2.9}$$

6. Compute the inverse square root of the second moment matrix at $\mathbf{x}_w^{(k)}$: $\mu_i^{(k)} = \mu^{-\frac{1}{2}}(\mathbf{x}_w^{(k)}, {\sigma_I}^{(k)}, {\sigma_D}^{(k)})$.

7. Update the shape adaptation matrix to $U^{(k)} = \mu_i^{(k)} \cdot U^{(k-1)}$ and normalize $U^{(k)}$ so that $\lambda_{max}(U^{(k)}) = 1$.

8. Evaluate convergence and divergence,

$$\text{convergence ratio} = 1 - \frac{\lambda_{min}(\mu)}{\lambda_{max}(\mu)} < \epsilon_C \tag{2.10}$$

$$\text{divergence ratio} = \frac{\lambda_{max}(U)}{\lambda_{min}(U)} < \epsilon_D. \tag{2.11}$$

   The process converges when the matrix μ is sufficiently close to a rotation matrix, or equivalently, when its eigenvalues $\lambda_{max}(\mu)$ and $\lambda_{min}(\mu)$ are almost equal ($\epsilon_C \approx 0.05$). Divergence can be measured from the eigenvalues of U to discard features with deformations that are too eccentric (e.g., $\epsilon_D \approx 6$).

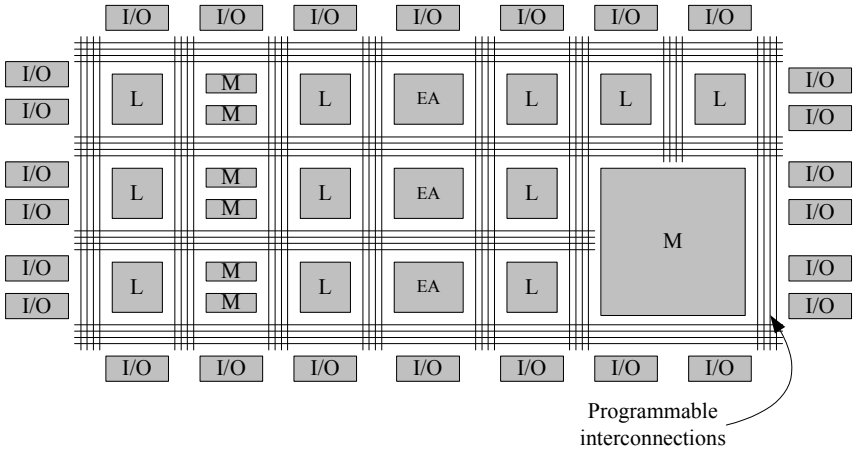


Figure 2.5: Structure of a generic FPGA.
The lines represent the reprogramable interconnections, L= programmable logic, M= memory block, EA= embedded arithmetic circuitry, I/O= I/O elements.

2.3 Field-programmable gate arrays and the Transmogrifier-4 (TM-4)

A field-programmable gate array (FPGA) is a semiconductor device that contains programmable logic components, programmable interconnects, memory blocks and I/O elements. Modern FPGAs may also include dedicated arithmetic circuitry, such as embedded multipliers and adders, and microprocessors either as fixed embedded components ('hard' processors) or software modules ('soft' processors). Figure 2.5 shows the structure of a generic FPGA.

The basic programmable unit of an FPGA is a *look-up table* (LUT). The latest families of FPGAs contain six-input LUTs, which can implement any basic logic function of up to six input variables (AND, OR, NOT, XOR, *etc.*). *Register* elements around the LUTs hold the values of the signals until some specified condition is met (generally a clock edge). LUTs and registers are combined into *logic blocks*[2] to

[2]In Altera's Stratix FPGAs, resources are quoted in terms of 'logic elements' (LEs). A logic element contains a 4-input look-up table, a flip-flop, a carry chain and routing logic. A detailed description of the architecture of Stratix devices can be found in [49].

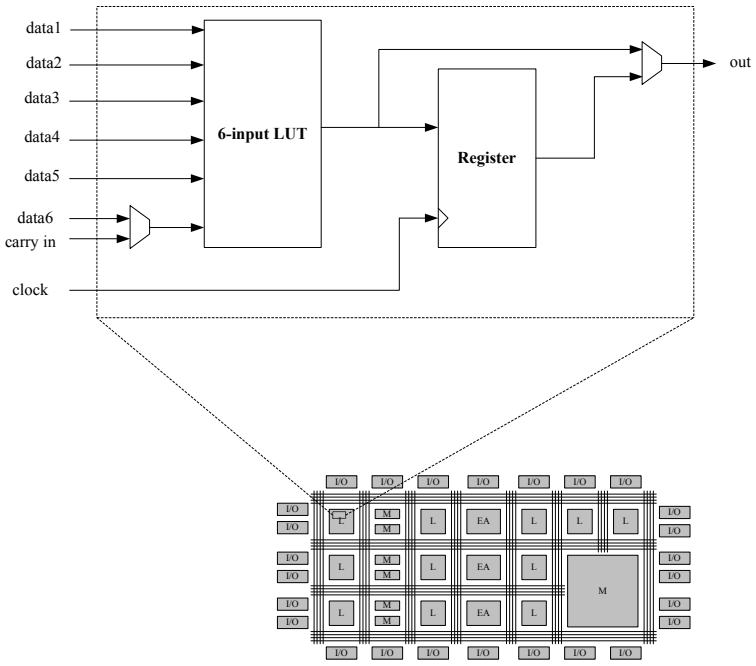


Figure 2.6: Generic logic block.

control both functionality and timing. Figure 2.6 shows the structure of a generic logic block. It includes multiplexers to choose between inputs and outputs, and a carry chain that can be used to cascade several blocks for implementation of fast adders and multipliers. Complex circuits can be designed by combining the functionality of multiple logic blocks. Thus, the processing power of an FPGA is directly related to the number of logic blocks it contains.

Hardware description languages such as VHDL and Verilog allow designers to specify the desired behaviour of the circuit in terms of basic logic functions, or more elaborate operations such as counters and multipliers. In addition, FPGA vendors offer intellectual property (IP) cores that implement commonly used functions, and that are optimized for their target FPGA architectures. FPGA vendors also offer software tools that can analyze the description of the circuit, decide on the optimal routing and allocation of resources, and translate this information into a bit-stream that can be downloaded onto the chip to configure it.

The Transmogrifier-4 is a prototyping platform developed at the University of Toronto [51]. It contains four Altera Stratix EP1S80F1508C6 FPGA devices, connected to each other through point-to-point buses. Each of the FPGAs is also connected to two 1GB DDR SDRAM banks, an I/O connector and a development bus that allows for transfer of download and control signals between the FPGAs and an on-board computer. NTSC and firewire video input and VGA output interfaces are connected to one of the FPGAs. The TM-4 includes two globally synchronous clocks that can be programmed independently.

The TM-4 uses the *ports* package [17], which allows communication between a UNIX terminal and the TM-4's on-board computer. Information can be transferred to and from a circuit running on the FPGAs by connecting signals in the circuit to a small *portmux* circuit created by the *ports* package.

2.4 Other vision systems on FPGAs

FPGAs are increasingly being used for feature extraction and other vision tasks to improve the performance of real-time systems. The most recent FPGA-based systems implement tasks such as stereo disparity estimation [11, 33], optic flow computation [44], template matching [25] and gesture recognition [47, 41], among others.

Among the FPGA-based feature detectors in the literature, many implement Tomasi and Kanade's corner detector [52] (similar to the Harris detector) as an initial stage for tracking [4, 15, 18, 5]. This detector is robust to rotation and translation, but not to changes in scale. Benedetti and Perona [4] implemented this algorithm and simplified the eigenvalue computation to avoid the use of square root operators. Their system uses a Xilinx XC4000 FPGA and is capable of processing NTSC video at 30 frames per second. Giacon *et al.* [18] built a similar circuit in a Xilinx Spartan 3 FPGA. The detector is capable of processing 512×512 images at 100 frames per second at a single scale. Bissacco and Ghiasi [5] implemented the algorithm in a

Xilinx Virtex Wildstar/PCI FPGA board. Feature selection takes 160 microseconds for a 640 × 480 image at 100 MHz.

Se *et al.* [46] use the scale-invariant SIFT features [30] for localization and terrain modelling in the ExoMars Rover. Their system can compute SIFT locations and descriptors for a 640 × 480 image in 60 ms using a Xilinx Virtex II FPGA.

Bouganis *et al.* [6] compute corner locations using complex steerable wavelets at four different scales and orientations. The detection algorithm is invariant to scale changes and

In their design, only one scale is implemented in hardware and is reused to compute the other scales, which allows them to fit the system in one Virtex V1000 FPGA. The system achieves a speed up factor of 8 over the software implementation.

To the knowledge of the author, there is no FPGA system in the literature that implements Mikolajczyk and Schmid's Harris-Affine detector. Dellaert and Tariq [12] introduced a multi-camera pose tracker that finds point correspondences in different camera views using affine-invariant features, however, at the date of publication the FPGA implementation of the feature detector was still in progress.

The iterative nature of the Harris-Affine detector makes it specially challenging to implement in hardware, due to the significant amount of resources required to implement each iteration and because of the numerical precision necessary to achieve convergence.

2.5 Design methodology

The first stage of the design process was to become familiar with the capabilities of the target platform, the Transmogrifier-4. For this purpose, a previously developed hardware implementation of a Canny edge detector [24] was ported to the TM-4. Details of this implementation are available in Appendix D. The Video Input and Video Output modules to be described in Section 3.2 were originally designed and

tested for this Canny detector circuit.

The design of the feature detector itself started by dividing the algorithm into functional modules and implementing them in MATLAB using floating-point arithmetic. This provided valuable information on the types of operations involved in the algorithm, as well as the dynamic range and precision of the parameters and intermediate results.

Once the floating point implementation was tested and verified, a fixed-point version of the algorithm was derived from the floating-point implementation by limiting the range and precision of all intermediate results to an explicit number of integer and fractional bits. This fixed-point implementation was used to study the effect of fixed-point arithmetic on the algorithm, to determine which parameters and intermediate results affected the final results most significantly, and to select the optimal number of bits to represent each quantity. This information was used to design the hardware architecture and to code the description of the circuit in VHDL.

Once coded, each module in the hardware design underwent several stages of verification. First, the VHDL descriptions were simulated in ModelSim using specially designed testbenches to verify that all control signals and arithmetic results occurred at the correct clock cycles (cycle-true verification). Second, intermediate results from the ModelSim simulations were saved into text files and then imported into MATLAB, where they were compared to the corresponding results in the fixed-point implementation (bit-true verification). Finally, modules with critical timing requirements were tested on-chip using the SignalTapII Logic Analyzer provided by Altera, or by saving the results into on-chip RAM and reading them through the *ports* package available in the Transmogrifier-4.

In parallel with the verification process, the VHDL descriptions of the individual modules were compiled with the Quartus II software from Altera, to obtain initial estimates of resource utilization and timing constraints. These estimates were used to improve the performance of the circuit and to allocate the available hardware

resources more efficiently.

Verification of the complete integrated system followed a similar procedure to the one outlined above. The final verification stages consisted of comparing the results from the board (retrieved through the ports package) with the fixed-point MATLAB implementation to discard the possibility of erroneous results due to timing glitches in the circuit. In addition, the VGA output served as an extra source of information to verify timing.

The development tools used in this project were ModelSim SE 6.0b [39], Quartus II 5.1 [42] and MATLAB 6.5.0 [50]. An Altera Development and Education Board (DE2) and Xilinx ISE6.3 were also used for the implementation of the Canny edge detector.

2.6 Summary

This chapter introduces some of the concepts applied throughout this thesis, presents a brief literature review on feature detection algorithms and FPGA-based vision systems, and discusses the design methodology applied in the implementation of the hardware system.

Features are points or regions of interest in an image. There are numerous approaches to feature detection, however they all attempt to achieve invariance to viewing conditions to facilitate the description and matching of objects across images. Modern feature detection algorithms are robust to rotation and translation, as well as to changes in scale and affine transformations.

The system described in this thesis is an FPGA-based implementation of the Harris-Affine feature detector introduced by by Mikolajczyk and Schmid [36, 37]. The detector combines a scale-adapted version of the Harris corner detector, with an iterative procedure that refines the location, scale and neighbourhood of a feature to achieve invariance to affine transformations.

The system is implemented on the Transmogrifier-4 (TM-4), a prototype platform that includes four Altera Stratix FPGAs and video I/O interfaces. The use of FPGAs is expected to increase the speed of the feature detector considerably, at the expense of loss of numerical precision. Chapter 3 provides a detailed description of the design of the hardware implementation.

Chapter 3

Hardware design

The algorithm for the scale and affine invariant interest point detector presented in Chapter 2 relies to a great extent on the accuracy of intermediate results to achieve convergence of the location, scale and shape matrix of a feature point. The main challenge when implementing such a complex algorithm in hardware is to balance the need for numerical precision with an efficient use of the available hardware resources. These resources include programmable logic in the form of look-up-tables, dedicated arithmetic circuitry such as embedded multipliers and adders, on-chip and off-chip memory capacity, and inter-chip communication bandwidth, among others.

Another important consideration in the design of the system is that the amount of data that needs to be processed by the iterative portion of the algorithm can vary significantly depending on the nature of the images being processed and the value of the input parameters. The number of preliminary features detected by the multiscale Harris stage depends on the amount of detail in the image, the base scale σ_{N0}, the scale factors s and ε and the threshold on the *cornerness* function. Because the iterative portion of the algorithm processes only preliminary features, the time it takes to process a single video frame depends on the number of features detected and how fast these converge. This poses a challenge from the hardware perspective because the system has to be capable of handling both very long and very short

delays. Delays play an important role when detecting new frames and when merging results from the system with the video stream for video output.

This chapter describes the implementation of the interest point detector on the TM-4 board and presents the reasoning behind the most important design decisions. Section 3.1 provides a high-level overview of the system. Section 3.2 gives a brief description of the input and output modules that interface with the NTSC camera and the VGA monitor. Sections 3.3 and 3.4 provide detailed descriptions of the core modules of the feature detector. Section 3.5 describes the distribution of the completed system on the TM-4. Finally, Section 3.6 discusses the methods and difficulties in translating an algorithm designed for floating-point computation into a fixed-point design suitable for implementation on FPGAs.

3.1 System overview

Figure 3.1 shows the high-level architecture of the system. The Video Input module receives composite NTSC signals from a CCD camera, performs frame de-interlacing and color conversion and forwards grayscale pixel values to the processing stages of the feature detector.

The core of the detector consists of the same two main stages in the detection algorithm presented in Chapter 2. The first stage is a Multiscale Harris corner detector that provides candidate feature points at three scales specified by the user. The second stage refines the location, scale and shape matrix of the candidate features. The latter stage is equivalent to the iterative portion of the algorithm, with the exception that the loops have been unrolled into individual identical modules (One Iteration) to provide larger throughput.

First-in-first-out (FIFO) buffers store information about candidate feature points

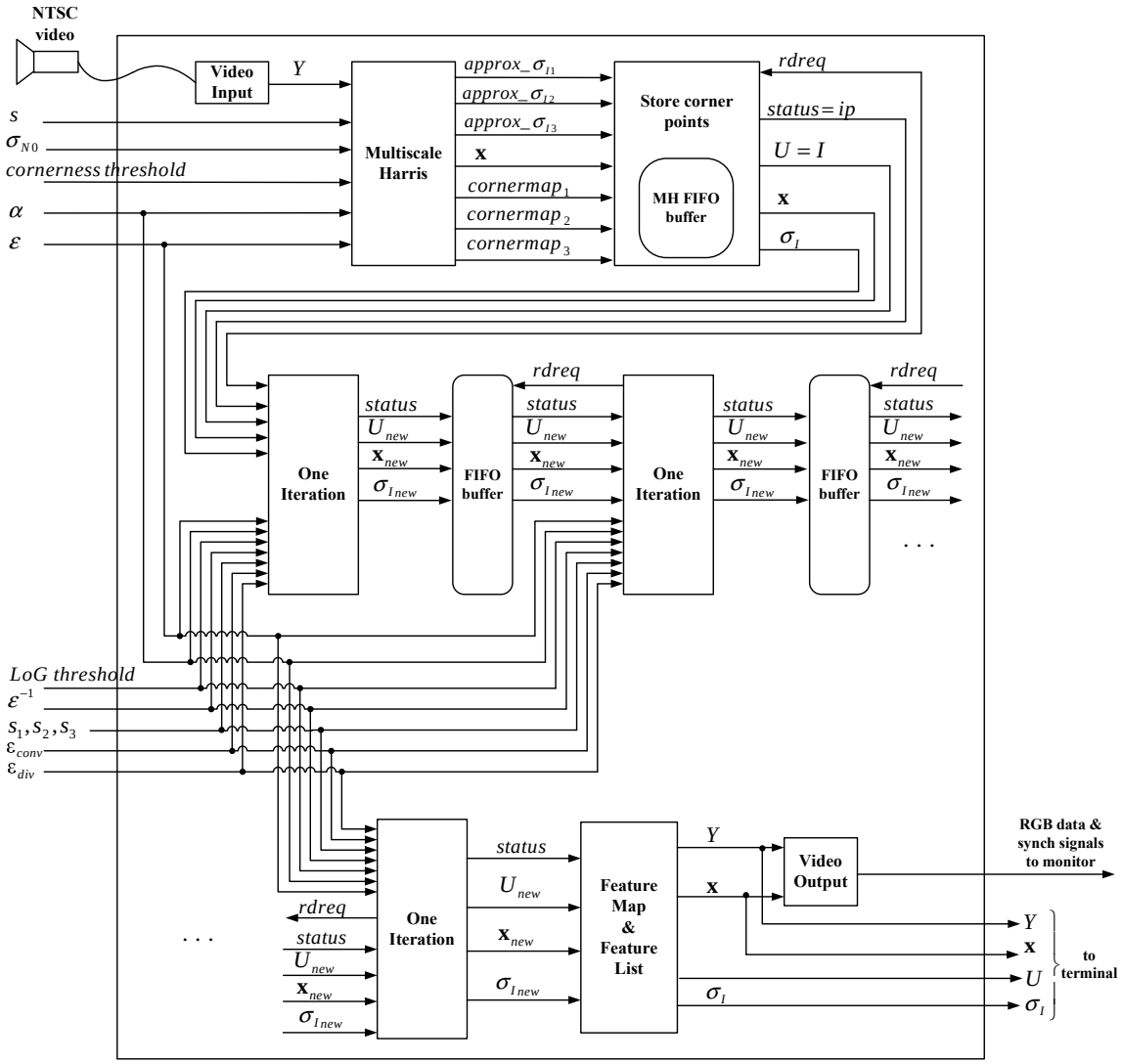


Figure 3.1: High-level architecture of the feature detector

after the Multiscale Harris module and after each iteration. Because preliminary feature points are discarded along the process, the FIFO buffers store only the information that corresponds to features that are either still being analyzed or have been accepted as final feature points. This shortens the processing latency of a frame by allowing the One Iteration modules to operate only on non-discarded feature points.

For this purpose, the One Iteration module includes two-bit *status* signals that indicate whether a feature is 'in progress', 'discarded' or 'accepted'. The functionality of most processing stages in the One Iteration module depends on the current status of a feature. For example, a feature marked as 'accepted' is kept in the pipeline of iterations until such time when the results from the system are written to the final feature map, however its location, scale and shape matrix are not processed any further.

One of the considerations in the design of the detector was to allow for scalability. For example, whenever possible the bit-widths of the parameters and intermediate signals have been defined as generic parameters, to allow for easier modification in case a different precision is required. As well, modules that operate on several scales in parallel consist of identical structures that are "stacked", so that more scales can be added with as few modifications as possible if more hardware resources become available.

It is also worth noting that a single One Iteration module could be used to repeatedly process features by feeding the results of one pass through the module back into the module. This limits the number of features that can be processed in a certain period of time, but allows for scalability in cases where there are not enough resources to pipeline several iteration blocks.

3.2 Video Input and Video Output modules

The Video Input and Video Output modules provide interfaces between the processing stages of the feature detector and the camera and monitor. These modules were adapted from the video and memory interfaces in [14], available as a TM-4 reference design.

As mentioned previously, the Video Input module receives composite NTSC video signals from the camera. A video processor chip on board the TM-4 translates these analog signals into digital synchronization signals and RGB values corresponding to pixels in a frame. The RGB values arrive in raster-scan order at an average rate of approximately 10.4 MHz (480 scanlines $\times$ 720 columns $\times$ 30 frames = 10368000 pixels per second), and are written to a frame buffer implemented in one of the external DDR SDRAM banks connected to FPGA 0.

Since the NTSC camera provides interlaced video (an image frame is transmitted as an even field formed by the even-numbered scanlines followed by odd field formed by the odd-numbered scanlines), the Video Input module performs de-interlacing by weaving the fields while they are being written to the buffer.

When a new pixel is needed, the corresponding RGB value is read from the frame buffer, its color is converted to YCrCb and the Y component is forwarded to the processing stages of the detector. In this implementation, the feature detector processes a 640×480 grayscale image at a rate of 12 MHz.

At the back end of the system, the Video Output module stores the results from the processing stages in a second frame buffer (implemented in the second SDRAM memory bank connected to FPGA 0). The module generates synchronization signals and reads the results from the buffer at the monitor clock rate (25 MHz). These digital signals are then transformed into the analog signals that drive the monitor by a video digital-to-analog converter.

It is worth noting that the frame buffers used in the Video Input and Video Output modules separate the clock domains of the camera, the monitor and the processing

stages. This makes it significantly easier to experiment with asynchronous designs, different processing speeds and different monitor resolutions.

As well, although the detector only processes the Y component of the color, the Video Input and Video Output modules have been designed to transfer and store RGB data to allow the same video interfaces to be used with other processing stages that may require RGB data. This results in larger memory requirements, but makes the architecture of the system more general.

3.3 Multiscale Harris module

The Multiscale Harris module computes the location of Harris corners at three scales simultaneously as depicted in Figure 3.2.

The three target integration scales (σ_{I1}, σ_{I2}, σ_{I3}) and their associated differentiation scales (σ_{D1}, σ_{D2}, σ_{D3}) are computed from the user-specified parameters σ_{N0}, ε and s, which represent, respectively, the base integration scale, the step between integration scales, and the factor between an integration scale and its associated differentiation scale. These parameters are related to the scales by the following equations:

$$\begin{aligned} \sigma_{I1} &= \sigma_{N0} \\ \sigma_{I2} &= \sigma_{N0} \cdot \varepsilon \\ \sigma_{I3} &= \sigma_{N0} \cdot \varepsilon^2 \\ \sigma_{D1} &= s \cdot \sigma_{I1} \\ \sigma_{D2} &= s \cdot \sigma_{I2} \\ \sigma_{D3} &= s \cdot \sigma_{I3} \end{aligned} \tag{3.1}$$

The optimal values for σ_{N0}, ε and s are application dependent, as is the number of scales that need to be analyzed to find a significant number of stable feature points. The authors of [37] suggest a value of ε between 1.1 and 1.4 and a value of s of 0.7,

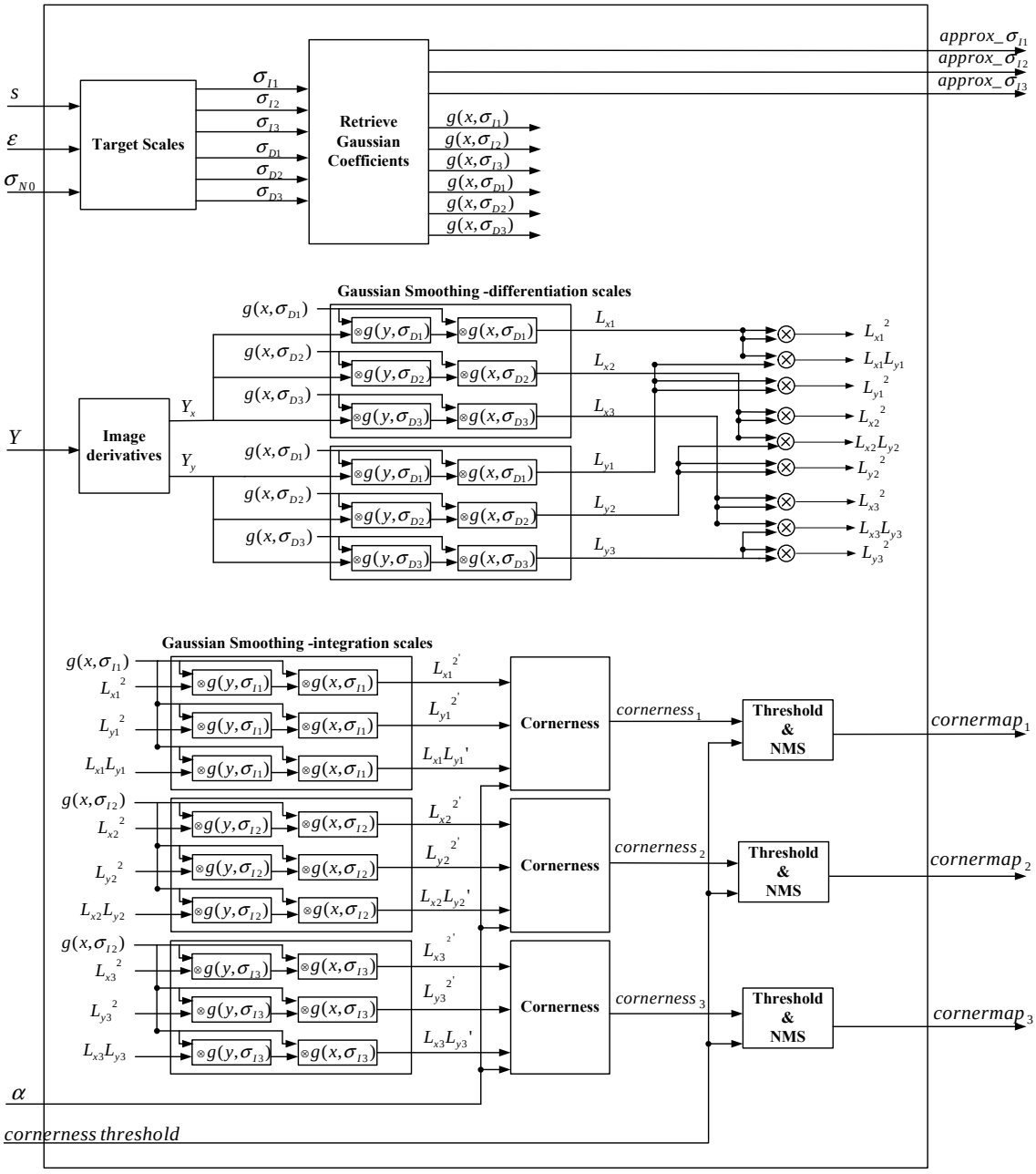


Figure 3.2: Multiscale Harris module

but the base scale and the number of scales are left to the user's discretion. Although in general it is best to analyze as many scales as possible, this comes at the expense of extra hardware resources. For this reason, in this implementation of the multiscale Harris algorithm, each scale analyzed uses its own set of filters, buffers and other processing units to facilitate adding extra scales if more resources become available.

Each integration and differentiation scale is used to retrieve a set of Gaussian coefficients from a look-up table in memory. These coefficients form an 11-tap filter at the given scale that is used to perform smoothing of intermediate results. A detailed explanation of the Gaussian look-up table and the retrieval process is provided in Section 3.3.1.

The first image processing stage in the module computes the derivatives of the input image in the x- and y-directions by filtering the input pixel stream with the central-difference kernels

$$\frac{d}{dx} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \frac{d}{dy} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{3.2}$$

The derivatives, Y_x and Y_y in Figure 3.2, are then convolved with Gaussian kernels at the differentiation scales σ_{D1}, σ_{D2} and σ_{D3}, first along the y-direction and then along the x-direction. The implementation of the filtering operations is described in detail in Section 3.3.2.

The results from the convolutions are combined to create the products ${L_{xi}}^2$, $L_{xi}L_{yi}$ and ${L_{yi}}^2$, $i = 1, 2, 3$. These correspond to the elements ${L_x}^2$, L_xL_y and ${L_y}^2$ of the second moment matrix μ discussed in Chapter 2:

$$\mu(\mathbf{x}, \sigma_I, \sigma_D) = {\sigma_D}^2\, g(\sigma_I) \otimes \begin{bmatrix} {L_x}^2(\mathbf{x}, \sigma_D) & L_xL_y(\mathbf{x}, \sigma_D) \\ L_xL_y(\mathbf{x}, \sigma_D) & {L_y}^2(\mathbf{x}, \sigma_D) \end{bmatrix}. \tag{3.3}$$

The products are convolved with Gaussian kernels at the corresponding integration scales σ_{Ii} and the results are used to evaluate the *cornerness* function at each scale

pair $(\sigma_{Ii}, \sigma_{Di})$:

$$cornerness = \det(\mu(\mathbf{x}, \sigma_I, \sigma_D)) - \alpha \operatorname{trace}^2 \mu(\mathbf{x}, \sigma_I, \sigma_D). \tag{3.4}$$

A threshold is applied to reject feature points with small *cornerness* (and thus less stable to changes in imaging conditions [37]). The selected preliminary feature points are those whose *cornerness* achieves a local maximum in their 8-point neighbourhoods.

The module outputs the pixel locations ($\mathbf{x}$), the three approximate integration scales ($approx\_\sigma_{I1}$, $approx\_\sigma_{I2}$ and $approx\_\sigma_{I3}$), and three 'corner' maps ($cornermap_1$, $cornermap_2$ and $cornermap_3$) that indicate whether or not the pixel was chosen as a feature at each of the integration scales. This information is passed on to the following module, Store corner points, where the location and scale of the corner pixels are stored in the MH (Multiscale Harris) FIFO buffer.

The pipelined architecture of the Multiscale Harris module is capable of producing the corner maps for a pixel in the input image every clock cycle at a rate of 45 MHz. In the current implementation, the module runs at 12 MHz because of timing requirements in other parts of the system.

3.3.1 Retrieve Coefficients module

In a floating-point implementation, the Gaussian coefficients needed for a smoothing filter would be obtained by evaluating Equation 3.5 at each scale σ and coordinates (x, y) in the filter,

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(\frac{-(x^2 + y^2)}{2\sigma^2}\right). \tag{3.5}$$

In a fixed-point hardware implementation, however, the evaluation of the exponential and the division operations are expensive in terms of resources and may yield inexact results.

This implementation uses a different approach, which consists of creating a look-up table of scales and their associated Gaussian coefficients and referring to it whenever

a new Gaussian kernel is needed. The table can be created in software and saved as a text file. A small C program then reads from the file and transfers the data to the TM-4, where the circuit loads the table into on-chip RAM.

Given a target scale σ, the Retrieve Coefficients module searches the look-up table for the closest scale available and returns this approximate scale and its associated Gaussian coefficients.

The number of scales in the table is only limited by the amount of on-chip RAM available and the delay that can be tolerated (it takes one clock cycle to compare the target scale with each scale in the table). In the current implementation, the table contains data for 60 scales, ranging between $\sigma = 0.75$ pixels and $\sigma = 8$ pixels.

The user is free to change the scales in the table to fit the application, as long as some conditions are met. First, the scales have to be kept in ascending order. Second, there are some restrictions on the range and precision of the scales that can be included in the table because of the fixed number of bits allocated to represent them. Section 3.6 discusses the choices of bit-widths and their effect on the system's performance.

3.3.2 Image derivatives and Gaussian filtering

Filtering operations, whether for image differentiation or image smoothing, form the backbone of the image processing stages in the feature detector. Moreover, they use a significant portion of the hardware resources allocated to the system. For these reasons, it is important to design efficient filtering architectures to improve the performance of the system as a whole.

The architecture of a finite-impulse response (FIR) filter, like the ones implemented in this system, consists of two main parts: an arithmetic component composed of adders and multipliers, and a set of delays used to align the input pixel stream into the columns and rows of the image before the arithmetic operations.

The arithmetic component is significantly simplified by the use of Gaussian filters. Two-dimensional Gaussian filters are separable, that is, their impulse response $g(x, y, \sigma)$ can be expressed as the product of two functions that depend on only one dimension:

$$\begin{aligned} g(x,y,\sigma) &= \frac{1}{2\pi\sigma^2}\ \exp\left(\frac{-(x^2+y^2)}{2\sigma^2}\right) \\ &= \frac{1}{\sqrt{2\pi\sigma^2}}\ \exp\left(\frac{-x^2}{2\sigma^2}\right) \times \frac{1}{\sqrt{2\pi\sigma^2}}\ \exp\left(\frac{-y^2}{2\sigma^2}\right) \\ &= g(x,\sigma) \times g(y,\sigma). \end{aligned} \tag{3.6}$$

In filtering, this means that convolving an image with an $N \times N$ Gaussian kernel $g(x, y, \sigma)$ is equivalent to convolving the image with the $1 \times N$ kernel $g(x, \sigma)$ along the horizontal x-direction, and then convolving the result with the $N \times 1$ kernel $g(y, \sigma)$ along the y-direction, or vice versa:

$$\begin{aligned} g(x,y,\sigma) \otimes I(x,y) &= g(x,\sigma) \otimes (g(y,\sigma) \otimes I(x,y)) \\ &= g(y,\sigma) \otimes (g(x,\sigma) \otimes I(x,y)) \end{aligned} \tag{3.7}$$

This separation reduces the complexity of the filter from $O(n^2)$ to $O(n)$, which translates directly into the number of multipliers that need to be instantiated in order to multiply all the coefficients of the kernel in parallel. Moreover, since the ranges of x and y are the same, the two one-dimensional kernels are equal.

In addition to being separable, Gaussian filters are also symmetric. A symmetric filter with coefficients $C_1, C_2, \ldots, C_N$, satisfies $C_1 = C_N$, $C_2 = C_{N-1}$, $C_3 = C_{N-2}$, etc. This quality results in two further simplifications of the architecture. First, it reduces the number of multipliers needed to implement the filter from N to $\frac{N-1}{2} + 1$ (for a filter with N odd), since the image pixels that have to be multiplied by the same coefficients can be added before the multiplication. Figure 3.3 shows the architecture of a general non-symmetric 11-tap filter, while Figure 3.4 illustrates the simplified structure in the case of a symmetric filter. Second, the look-up table of scales and Gaussian coefficients described in Section 3.3.1 only needs to store $\frac{N-1}{2}+1$ coefficients,

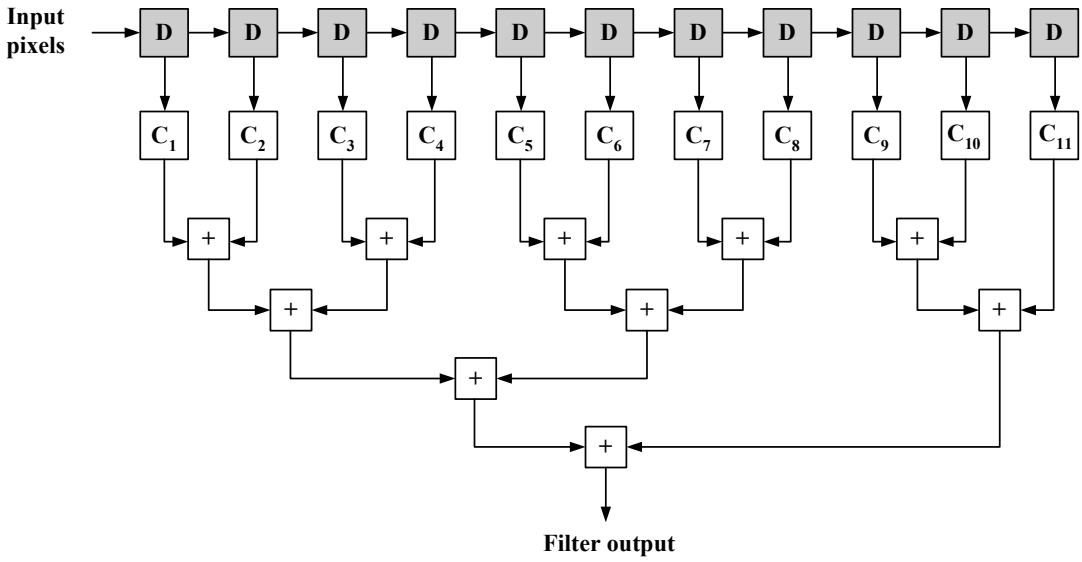


Figure 3.3: Structure of a non-symmetric 11-tap FIR filter

instead of all N. For an 11-tap filter with 8 bits per coefficient, this translates into a saving of 40 memory bits per scale.

Figure 3.5 shows a high-level diagram of a filter that includes the set of delays used to align the input pixels with the filter coefficients prior to the additions and multiplications. Because the pixels arrive in raster-scan order (row by row), $N-1$ scanlines have to be buffered in order to filter the image along the y-direction. For a 640×480 image and an 11-tap filter, this translates into $640 \times 10 = 6400$ delay elements in the y- delay buffer.

Filtering along the x-direction, on the other hand, only requires $N-1$ delay elements. For this reason, the image is filtered first in the y-direction and then in the x-direction, to allow the filtering operations at different scales (such as σ_{I1}, σ_{I2} and σ_{I3}) to share the same $y-$ delay buffer. This is justified by the fact that the order of the convolutions does not affect the final result, as shown in Equation 3.7.

A similar filter architecture is used for the computation of the image derivatives. The delay buffers hold two image scanlines and are shared between the 3×3 kernels $\frac{d}{dx}$

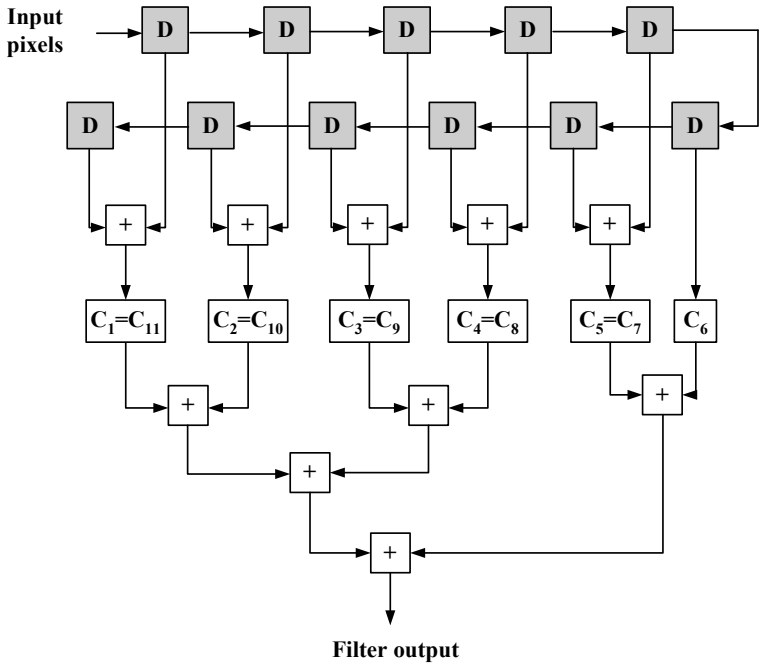


Figure 3.4: Structure of a symmetric 11-tap FIR filter

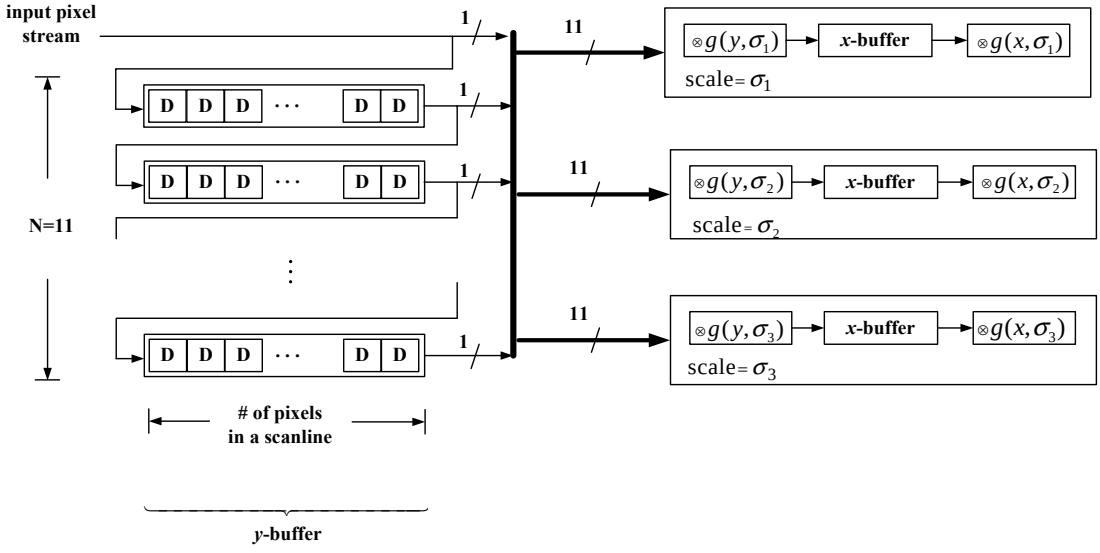


Figure 3.5: Architecture of a two-dimensional Gaussian filter.
Shift registers buffer the input pixels to align them with the filter coefficients prior to the additions and multiplications. The large y- buffer is shared among all scales.

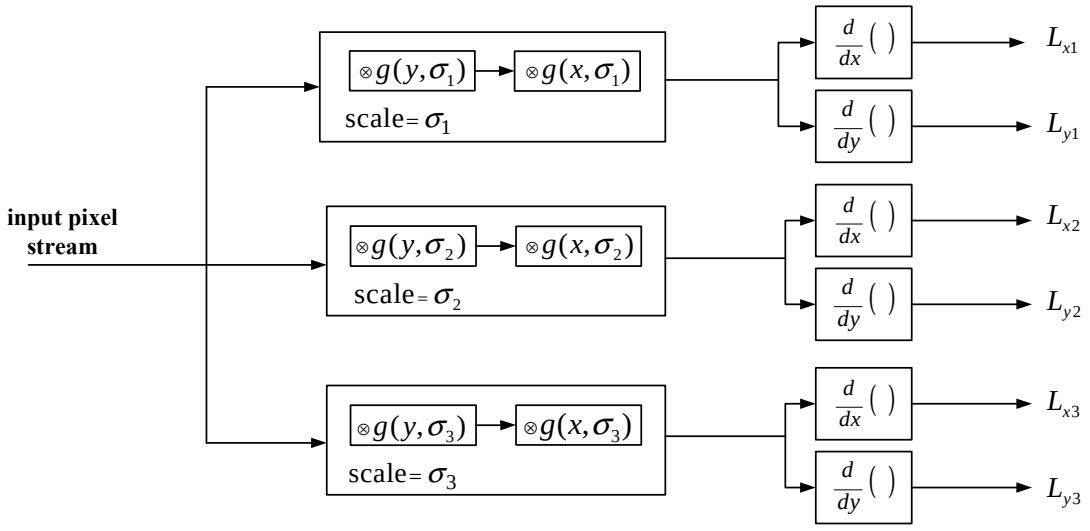


Figure 3.6: Convolution of an image with a Gaussian derivative filter. Gaussian smoothing at scales σ_{D1}, σ_{D2} and σ_{D3} is done before the differentiation.

and $\frac{d}{dy}$. The multiplications are done with bit-shift operators since all the coefficients can be expressed in terms of powers of 2.

The terms ${L_{xi}}^2$, $L_{xi}L_{yi}$ and ${L_{yi}}^2$, $i = 1, 2, 3$, are computed by convolving the image derivatives Y_x and Y_y with the Gaussian kernels at the differentiation scales σ_{D1}, σ_{D2} and σ_{D3}. As in the case of the convolutions with $g(x, \sigma)$ and $g(y, \sigma)$, the order of the operations (differentiation and smoothing) does not affect the results, but it does affect the number of scanlines that need to be buffered

If the smoothing is performed first, as show in Figure 3.6, only one Gaussian filtering operation is required per scale, and therefore the total number S of scanlines that have to be buffered for an $N \times N$ filter (excluding the small x-buffers) is equal to

$$S = \underbrace{N-1}_{y\text{-buffers}} \quad + \quad \underbrace{2 \times M}_{\text{derivative buffers}} \tag{3.8}$$

where M is the number of scales σ_D that are computed in parallel.

On the other hand, if the derivatives are computed first as shown in Figure 3.2,

the total number of scanlines buffered does not depend on the number of scales and is equal to $2 + 2 \times (N - 1)$.

For $N = 11$, the architecture in Figure 3.6 is more efficient than the one used in this implementation if $M \leq 5$. The Multiscale Harris module was implemented using the order of operations in Figure 3.2 because it is expected that more scales will be added to the module and it is preferable that the number of scanlines buffered remains constant for any number of scales.

3.4 One Iteration module

The purpose of the sequence of iteration modules is to refine the integration scale σ_I and the spatial location $\mathbf{x}$ of a candidate feature point produced by the Multiscale Harris module, and to generate a shape matrix U that describes the affine transformations performed on an image region around $\mathbf{x}$. Figure 3.7 shows the processing stages included in the One Iteration module.

The first step in an iteration is to retrieve the location, scale, status and shape matrix of a candidate feature point from a FIFO buffer. In the case of the first iteration, the FIFO contains the results from the Multiscale Harris module and therefore the input shape matrix is set to the identity matrix I and the status of the feature is set to 'in progress'.

As mentioned before, because the Multiscale Harris module is fully pipelined, it produces a result every clock cycle at a rate of 12 MHz. Only points that are selected as corner points are stored in the MH FIFO buffer, however the number of such points depends entirely on the characteristics of the scene being captured by the camera. Scenes that contain rich texture or a lot of detail are likely to have a large number of features.

Each One Iteration block, on the other hand, performs hundreds of memory accesses to process a single feature and thus there is a latency of several hundred clock

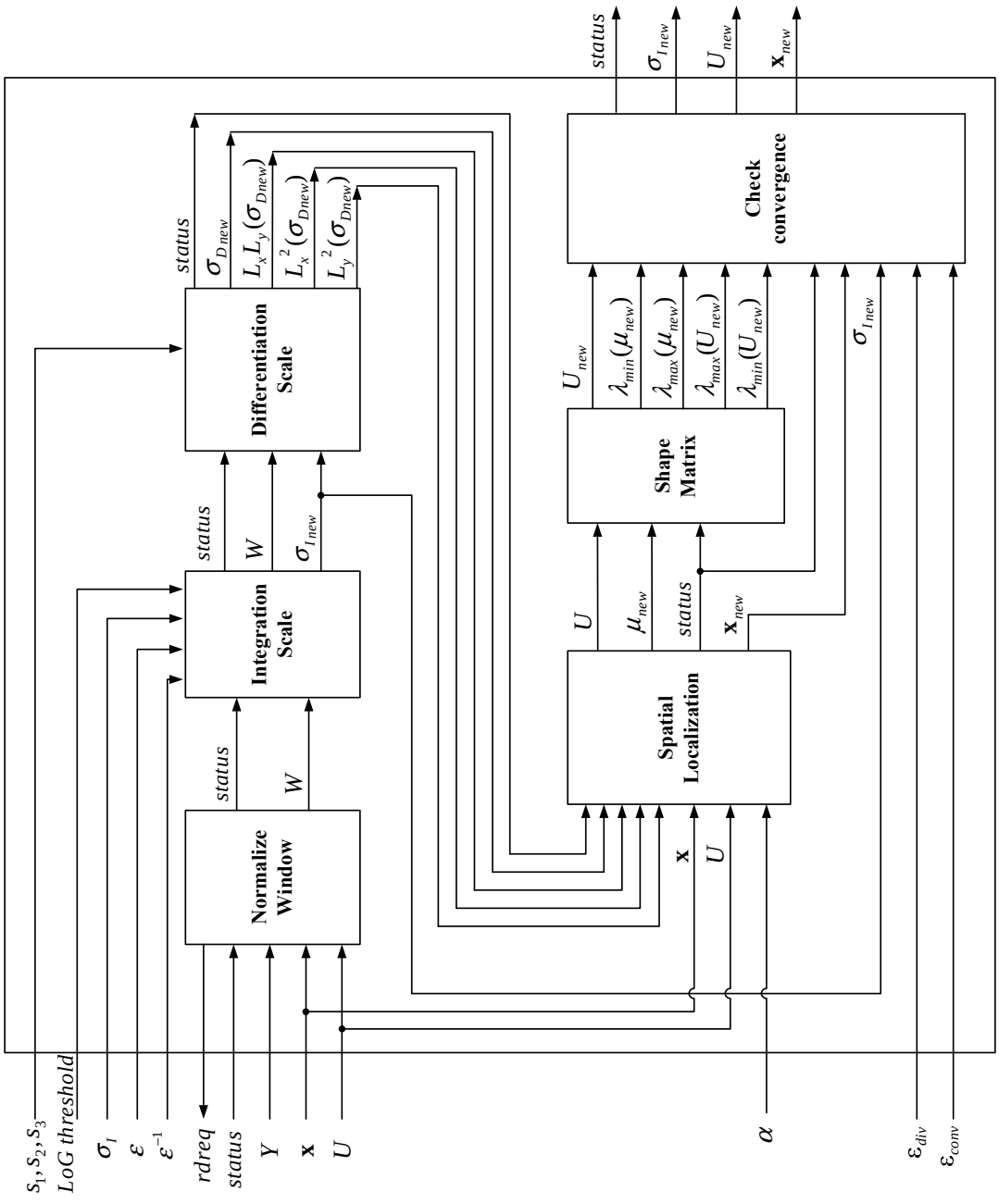


Figure 3.7: One Iteration module

cycles between the time consecutive corner points are retrieved from the MH FIFO. For these reasons, it is possible for the Multiscale Harris module to produce more candidate feature points per image frame than a One Iteration module can process. If this happens, the MH FIFO buffer will fill up and any incoming features will be dropped.

To minimize the number of candidate features lost, the sequence of iteration modules runs at five times the speed of the Multiscale Harris module, or equivalently at a clock rate of 60 MHz. Moreover, the internal architecture of the iteration block is almost entirely pipelined, which allows it to process a distinct feature in each of its arithmetic modules at any point in time.

3.4.1 Normalize Window module

As seen in Chapter 2, the Harris-Affine detector achieves invariance to affine transformations by performing all filtering, differentiation and comparison operations over affine-normalized, or warped, image regions. The main function of the Normalize Window module is to generate these warped image regions given the coordinates $\mathbf{x} = (x_{col}, x_{row})$ of a feature and its associated shape matrix U as calculated in the previous iteration. A schematic diagram of the module is shown in Figure 3.8.

The operations in this module are divided into three stages. The first one computes the coordinates $\mathbf{x}_w = (x_{w_{col}}, x_{w_{row}})$ of the feature in the transformed (warped) image domain. The second one calculates the coordinates of the pixels in a 25×25 pixel image patch W centered at $\mathbf{x}_w$ and performs an inverse mapping to match them to coordinates in the original image domain [57]. Finally, the grayscale values of the pixels in W are calculated through bilinear interpolation of neighbouring grayscale values Y in the original image, which are retrieved from a frame buffer implemented in on-chip RAM.

The size of the image patch W was chosen to minimize edge effects in an 8-point neighbourhood around $\mathbf{x}_w$. This neighbourhood is later used in the Spatial Localization module to update the location of the feature. The most noticeable edge effects are

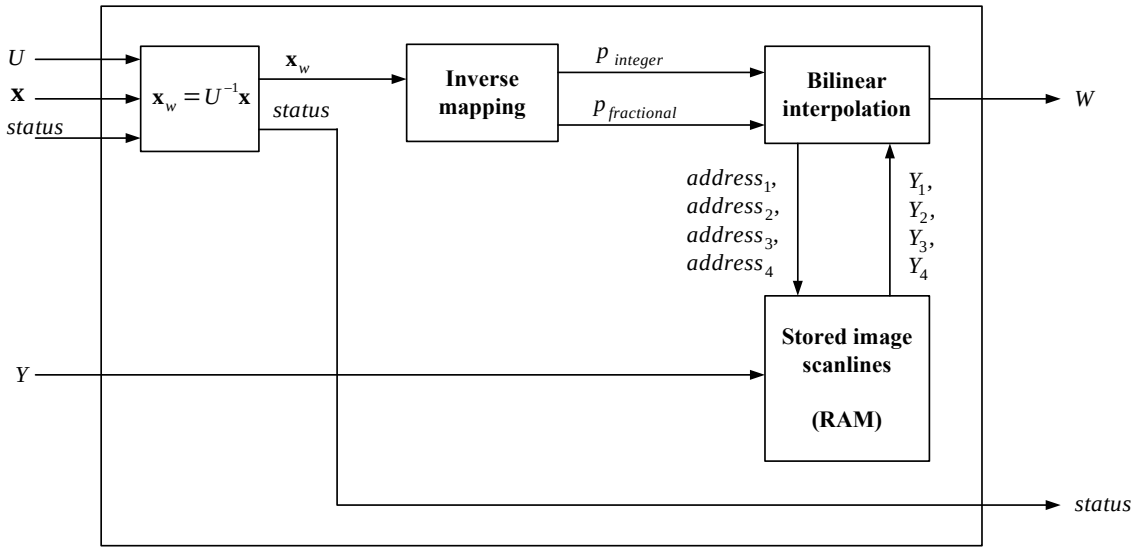


Figure 3.8: Normalize Window module

caused by the filtering operations required to calculate the differentiation scale, which consist of a convolution with a 3×3 kernel followed by two convolutions with 11×11 kernels. The number of pixels around the 8-point neighbourhood affected by these operations is 11, thus the minimum size of W must be $(11+3+11) \times (11+3+11) = 25 \times 25$ pixel^2.

In order to compute $\mathbf{x}_w$, we apply Cramer's rule (Appendix A) to the equation

$$\begin{aligned} U\mathbf{x}_w &= \mathbf{x} \\ \begin{pmatrix} U_{11} & U_{21} \\ U_{12} & U_{22} \end{pmatrix} \begin{pmatrix} x_{w_{col}} \\ x_{w_{row}} \end{pmatrix} &= \begin{pmatrix} x_{col} \\ x_{row} \end{pmatrix} \end{aligned} \tag{3.9}$$

which gives

$$x_{w_{col}} = \frac{x_{col}\,U_{22} - x_{row}\,U_{21}}{U_{11}\,U_{22} - U_{21}\,U_{12}} \qquad \text{and} \qquad x_{w_{row}} = \frac{x_{row}\,U_{11} - x_{col}\,U_{12}}{U_{11}\,U_{22} - U_{21}\,U_{12}}. \tag{3.10}$$

In the case where $\mathbf{x}_w$ lies outside the image, the status of the feature is updated from 'in progress' to 'discarded'.

The inverse mapping of the coordinates $\mathbf{p}_w$ of the pixels in the 25×25 warped image patch W to coordinates $\mathbf{p}$ in the original image domain is performed by multiplying

each coordinate $\mathbf{p}_w$ by the shape matrix U:

$$\mathbf{p} = \mathbf{p}_w \, U \tag{3.11}$$

Since in general the coordinates $\mathbf{p} = (p_{col}, p_{row})$ are not integers, the grayscale values in W are computed by bilinear interpolation of the four grayscale values closest to $\mathbf{p}$. The integer parts of $\mathbf{p}$ are used to retrieve the grayscale values from the RAM frame buffer, while the fractional parts are used as weights for the interpolation.

The need to retrieve four pixels from the frame buffer for every element in W causes a bottleneck in the system. In general, a dual-port RAM (two read/write ports) allows only one memory access per clock cycle (taking into account that one of the ports is permanently set to write since pixels are constantly being written to the frame buffer). For a 25×25 W, this translates into $25 \times 25 \times 4 = 2500$ clock cycles spent solely on memory accesses. A possible improvement that doubles the rate of memory access is to instantiate a three-port RAM buffer that allows simultaneous access to one write port and two read ports. However, this buffer requires two copies of the image frame and given the current image size of 640×480 pixels, it does not fit in the available on-chip RAM.

In practice, there are several approaches that take into account the relative locations of the data in memory. For example, since the four pixels required for the interpolation are in adjacent scanlines and adjacent columns, it is possible to access all four pixels at the same time by de-interlacing the incoming image along the scanlines and columns. For this purpose, the RAM frame buffer in this implementation consists of four smaller buffers. The first buffer stores pixels located at even scanlines and even columns, the second one stores pixels located at even scanlines and odd columns, the third one keeps pixels at odd scanlines and even columns and the last one keeps pixels at odd scanlines and odd columns. This approach results in a latency of $25 \times 25 = 625$ clock cycles instead of the original 2500, using the same amount of memory required for a single copy of the image frame.

Because of the size of W, the Normalize Window module requires that the frame

buffer hold at least 12 scanlines above and below $\mathbf{x}$. For this reason, the buffer is fed directly from the Video Input module to ensure that there are as many scanlines as possible stored in the buffer by the time the first feature is retrieved from the MH FIFO buffer. Given that the Multiscale Harris module has a latency of approximately 7700 clock cycles, there will be 12 scanlines by the time the first preliminary feature arrives to the module.

3.4.2 Integration Scale module

As mentioned in Chapter 2, the initial estimate of the integration scale provided by the scale-adapted Harris detector needs to be refined to account for affine transformations in the image. The new integration scale at each iteration, $\sigma_{I_{new}}$, is the scale at which the absolute value of the normalized Laplacian (Equation 3.12) attains a maximum over scale,

$$|\mathrm{LoG}(\mathbf{x}, \sigma_I)| = \sigma_I^2 \, |L_{xx}(\mathbf{x}, \sigma_I) + L_{yy}(\mathbf{x}, \sigma_I)|. \tag{3.12}$$

The algorithm in [37] calculates $\sigma_{I_{new}}$ at each iteration through a separate iterative process that updates the location and integration scale of a point until they converge to steady values. The hardware resources available for this implementation however are not sufficient to implement an iterative process within each One Iteration module. For this reason, we use the alternative method outlined in [37] in which the Laplacian is computed at the original scale σ_I and at the immediate finer (σ_{I2}) and coarser (σ_{I3}) scales. $\sigma_{I_{new}}$ is set to the scale that achieves a maximum value of the Laplacian, without further refinement of the location.

The Integration Scale module, shown in Figure 3.9, computes the Laplacian at the point $\mathbf{x}_w$ from the grayscale values in the image patch W. As a first step, the second derivatives of W along the x- and y- directions, W_{xx} and W_{yy}, are computed using the kernels that result from convolving the first derivative kernels $\frac{d}{dx}$ and $\frac{d}{dy}$

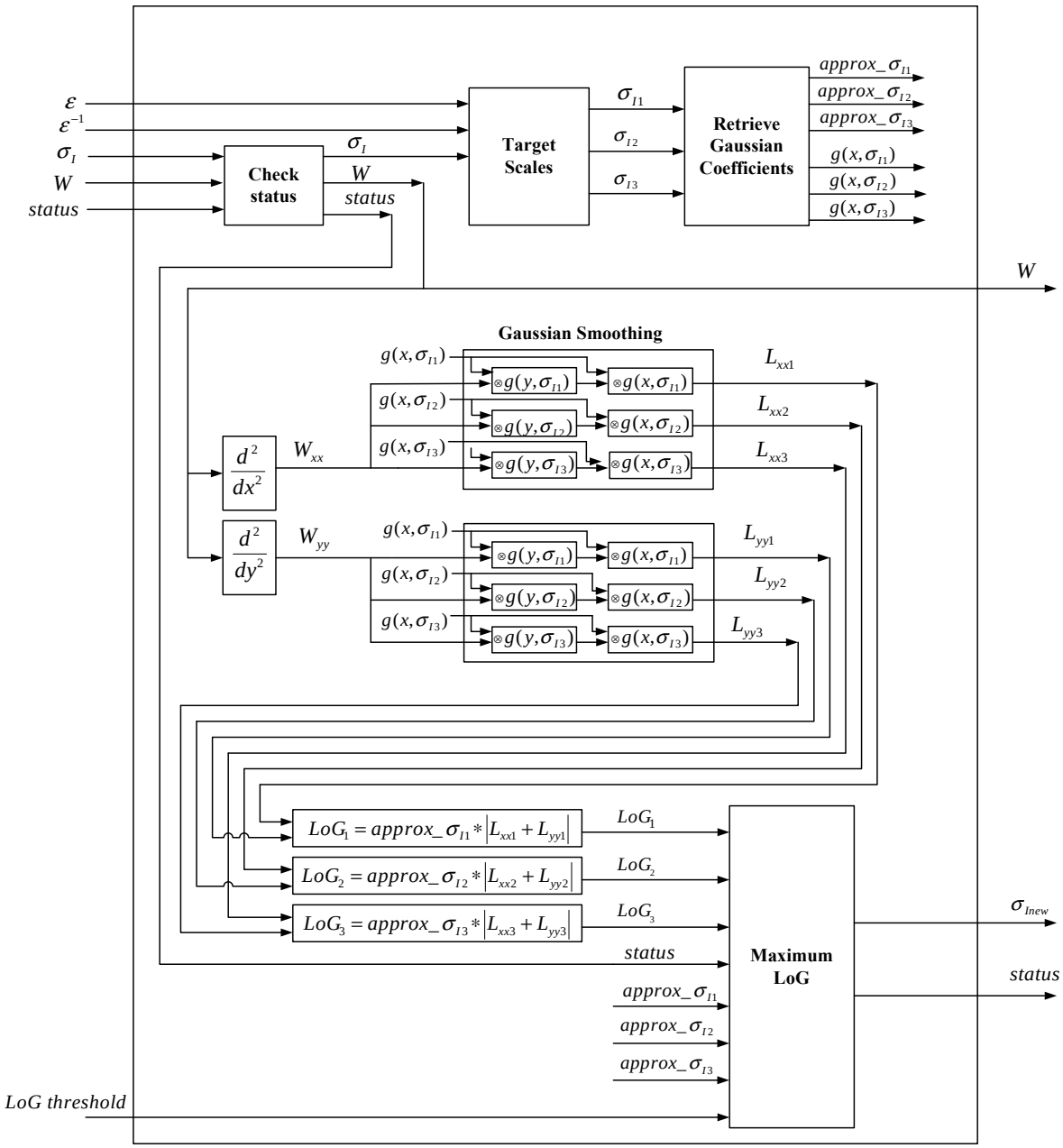


Figure 3.9: Integration Scale module

(Equation 3.2),

$$\frac{d^2}{dx^2} = \begin{bmatrix} 1 & 0 & -2 & 0 & 1 \\ 4 & 0 & -8 & 0 & 4 \\ 6 & 0 & -12 & 0 & 6 \\ 4 & 0 & -8 & 0 & 4 \\ 1 & 0 & -2 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \frac{d^2}{dy^2} = \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}. \tag{3.13}$$

The derivatives are then convolved by Gaussian kernels at each scale using the same procedure described in sections 3.3.2 and 3.3.1. Finally, the smoothed derivatives are used to evaluate the Laplacian, which in turn determines the value of $\sigma_{I_{new}}$. If the maximum value of the Laplacian falls below a user-specified threshold, $LoG\,threshold$, the feature is discarded.

3.4.3 Differentiation Scale module

Given the new integration scale σ_{Inew}, the system computes the differentiation scale σ_{Dnew} that maximizes the local isotropy ratio

$$Q = \frac{\lambda_{min}(\mu)}{\lambda_{max}(\mu)} \tag{3.14}$$

where $\lambda_{min}(\mu)$ and $\lambda_{max}(\mu)$ are respectively the minimum and maximum eigenvalues of the second moment matrix μ (Equation 3.3). The authors of [1] suggest searching for σ_{Dnew} among the differentiation scales $\sigma_{Dnew} = s \cdot \sigma_{Inew}$ obtained using values of s in the range $[0.5, 0.75]$.

The Differentiation Scale module (Figure 3.10) computes the second moment matrix μ for three differentiation scales in parallel. The factors s_i used to compute the scales can be selected by the user from any fractional number with up to eight bits of precision, and can be modified at run-time.

The steps followed to calculate the entries of μ, such as image differentiation and smoothing, are similar to those performed in the Multiscale Harris module, with the

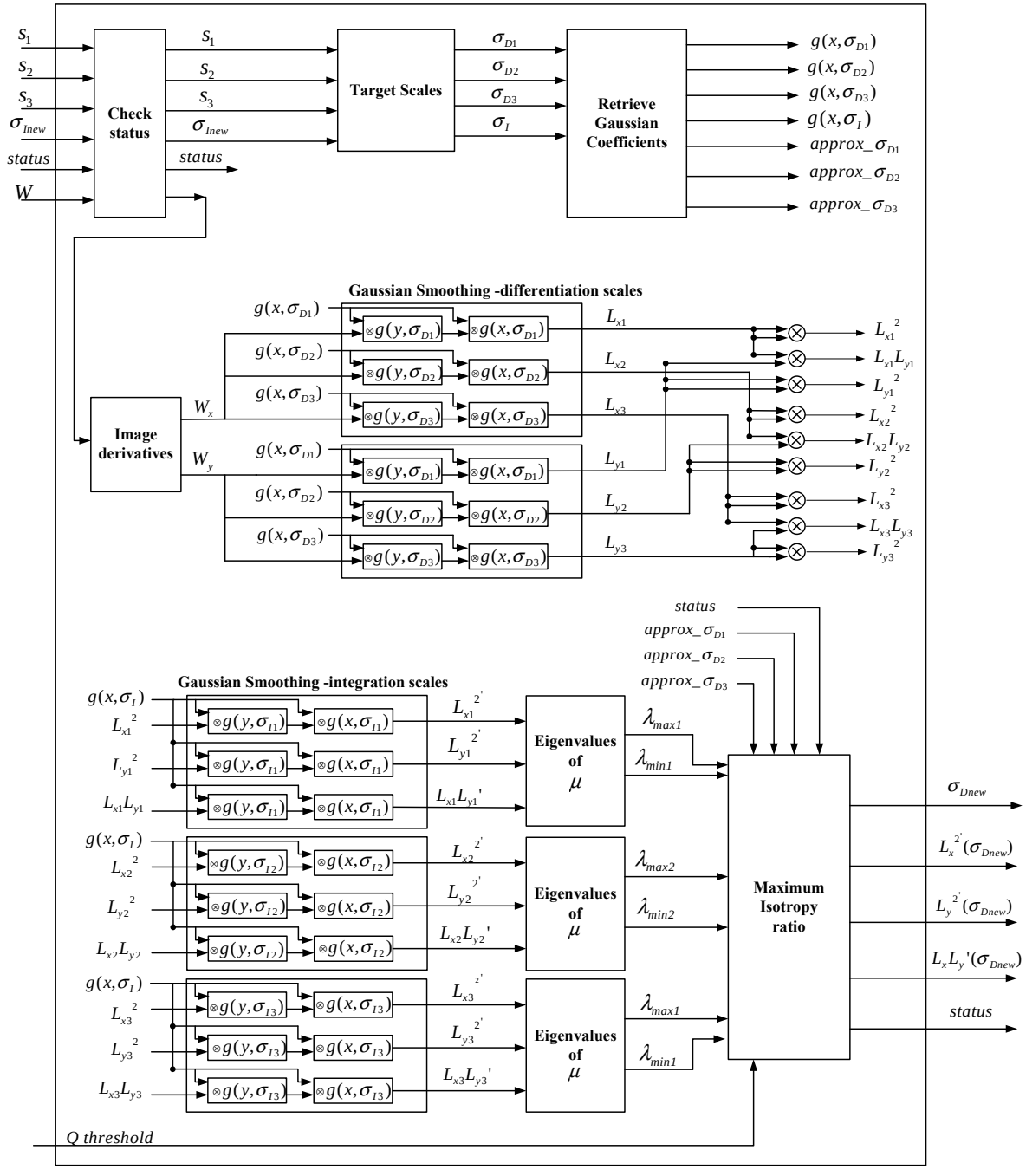


Figure 3.10: Differentiation Scale module

exception that the operations are performed over the image patch W instead of over the entire image.

The evaluation of Q requires the eigenvalues of μ. Because of the small size of the matrix (2×2), the module implements the direct algebraic solution to the eigenvalue problem, by which $\lambda_{min}(\mu)$ and $\lambda_{max}(\mu)$ are the roots of the characteristic equation

$$\det(\mu - \lambda I) = \det\left(\begin{bmatrix} {L_x}^{2'} & L_x {L_y}^{2'} \\ L_x {L_y}^{2'} & {L_y}^{2'} \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = 0 \qquad (3.15)$$

where the entries ${L_x}^{2'}$, ${L_y}^{2'}$ and $L_x {L_y}'$ correspond respectively to $g(\sigma_I) \otimes {L_x}^2(\mathbf{x}, \sigma_D)$, $g(\sigma_I) \otimes {L_y}^2(\mathbf{x}, \sigma_D)$ and $g(\sigma_I) \otimes L_x L_y(\mathbf{x}, \sigma_D)$ in Equation 3.3. This yields the following simplified expressions for the eigenvalues of μ (Appendix B):

$$\lambda_{min}(\mu) = \frac{1}{2}\left({L_x}^{2'} + {L_y}^{2'} - (|{L_x}^{2'} - {L_y}^{2'}| + 4\,|L_x {L_y}'|)\right)$$

and

$$\lambda_{max}(\mu) = \frac{1}{2}\left({L_x}^{2'} + {L_y}^{2'} + (|{L_x}^{2'} - {L_y}^{2'}| + 4\,|L_x {L_y}'|)\right).$$

Given the eigenvalues, we evaluate Q at each scale and determine which of them achieves the maximum isotropy ratio. The algorithm in [37] does not discuss the case in which two or more of the ratios are equal. In the case where the three ratios are the same, it is safe to assume that the function Q is flat in the range covered by the three scales, and thus any of the scales can be chosen. In the case where only two of the ratios are equal, the optimal scale could be found by fitting a smooth function to the values of Q at the three scales. This however, would involve recomputing the second moment matrix at the optimal scale, which would require extra time and hardware resources. Therefore, the module handles repeated value of Q by choosing the smallest differentiation scale among those with equal maximum ratio.

In addition, the module includes a threshold parameter that allows the user to discard features for which the ratio Q is too small. The threshold can be set to any fractional number with up to five bits of precision and can be modified at run-time.

The module outputs the chosen differentiation scale σ_{Dnew} and the status of the feature, as well as the elements of the second moment matrix at each point in a 3×3 region centered at $\mathbf{x}_w$, for the selected differentiation scale. The second moment matrices for the points around $\mathbf{x}_w$ are computed without any extra cost because of the pipelined architecture of the system. This data is readily available at the output of the module and saves the hardware resources that would have been needed to recompute the matrices in the Spatial Localization module.

3.4.4 Spatial Localization module

The second moment matrices from the Differentiation Scale module are used to evaluate the *cornerness* function at every point in the 3×3 region centered at $\mathbf{x}_w$. The relative location with respect to $\mathbf{x}_w$ of the pixel with the maximum value of *cornerness*, $\mathbf{x}_{max\,C}$, determines the new location $\mathbf{x}_{new}$ of the feature in the original image:

$$\mathbf{x}_{new} = \mathbf{x} + U \cdot \Delta\, x_w \tag{3.16}$$

where $\Delta\, x_w = \mathbf{x}_{max\,C} - \mathbf{x}_w$. It is worth noting that the actual value of $\mathbf{x}_w$ is not required to calculate $\Delta\, x_w$, since the relative position is measured with respect to the centre of the 3×3 region, regardless of its absolute location in the image.

In the case where $\mathbf{x}_{new}$ lies outside the boundaries of the image, the feature is discarded. The module (shown in Figure 3.11) outputs the new location $\mathbf{x}_{new}$ and the status of the feature, along with the second moment matrix that produced the maximum *cornerness* μ_{new}.

3.4.5 Shape Matrix and Check Convergence modules

Given the new second moment matrix, the Shape Matrix module (Figure 3.12) updates the shape adaptation matrix U by evaluating the equation

$$U_{new} = {\mu_{new}}^{-\frac{1}{2}} \cdot U \tag{3.17}$$

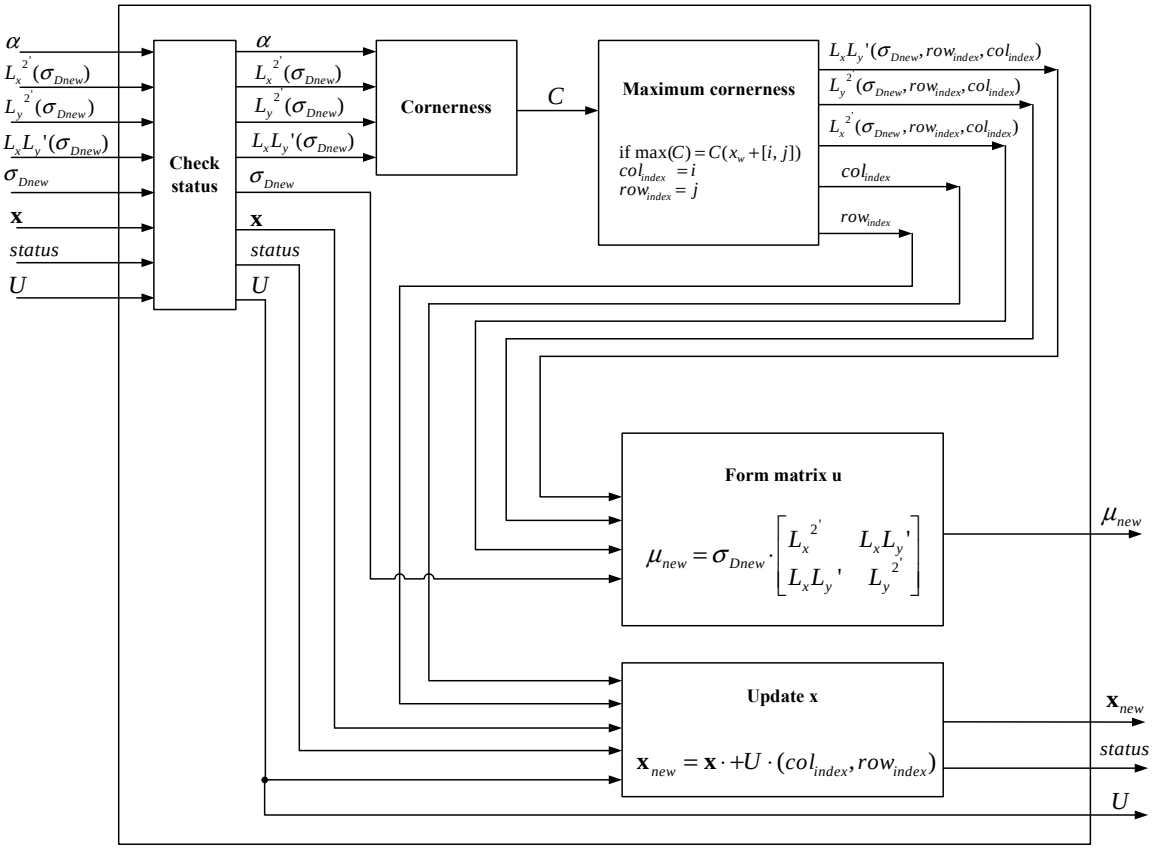


Figure 3.11: Spatial Localization module

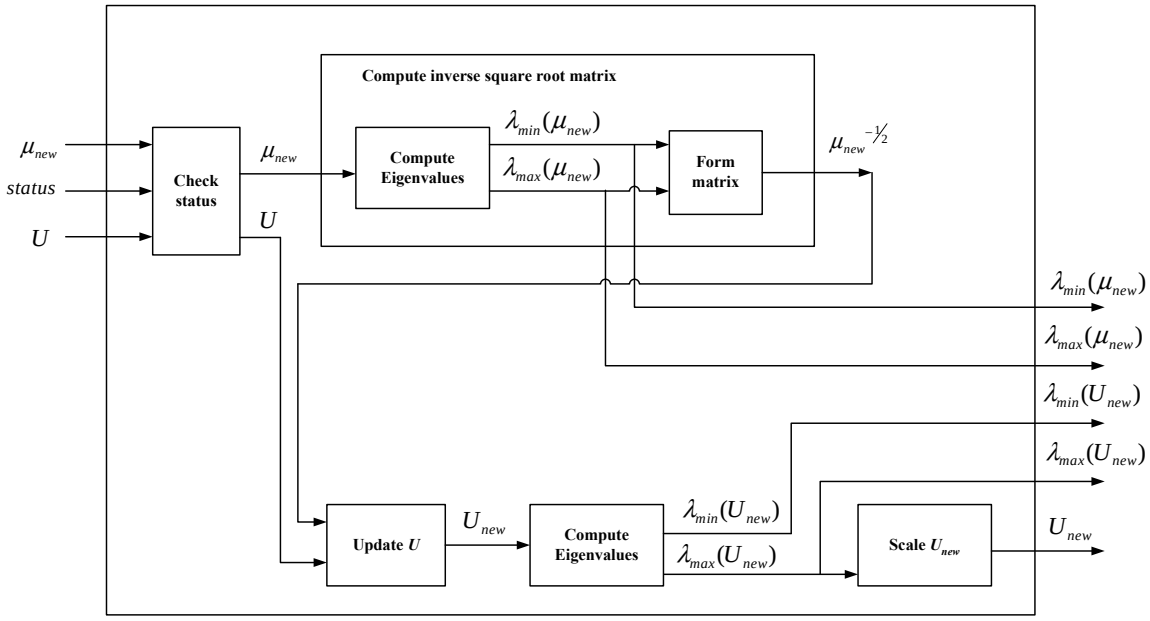


Figure 3.12: Shape Matrix module

where U_{new} is the updated shape matrix and ${\mu_{new}}^{-\frac{1}{2}}$ is the inverse square root of the second moment matrix. The elements of U_{new} are then divided by $\lambda_{max}(U_{new})$, so that the largest eigenvalue of the resulting matrix is normalized to one.

The steps followed to compute the inverse square root matrix ${\mu_{new}}^{-\frac{1}{2}}$ are explained in detail in Appendix C. The computation was greatly simplified by the fact that μ_{new} is a 2×2 symmetric matrix, which reduced the problem to a set of simple algebraic equations. Nonetheless, the computation of ${\mu_{new}}^{-\frac{1}{2}}$ is one of the most expensive processes in the implementation of the feature detector because of the use of very wide square roots, multipliers and dividers. To save resources, the operators were shared with other processes, such as the normalization of U_{new} and the computation of the convergence and divergence ratios.

The final step in the iteration computes a convergence ratio and a divergence ratio from the eigenvalues of μ_{new} and U_{new}:

$$\text{convergence ratio} = \frac{\lambda_{max}(\mu_{new})}{\lambda_{min}(\mu_{new})} \quad \text{and} \quad \text{divergence ratio} = \frac{\lambda_{max}(U_{new})}{\lambda_{min}(U_{new})} \tag{3.18}$$

The expression for the convergence ratio was modified from the original expression in [37] (Equation 2.10) to improve the accuracy of the result of the division. Thus, the relation between the original threshold, ϵ_c, and the modified threshold, ϵ_{conv} is

$$\epsilon_{conv} = \frac{1}{1 - \epsilon_c} \tag{3.19}$$

A feature is permanently accepted if its convergence ratio is below ϵ_{conv} and permanently discarded if its divergence ratio is greater than ϵ_{div}. The thresholds are specified by the user and can be modified at run-time. The number of bits allocated to these thresholds and other input parameters can be found in Table 3.1.

3.5 Feature detector on the Transmogrifier-4

As is common in hardware implementations, there is a tradeoff between the maximum speed the system can achieve and the amount of area (LUTs and other resources) it requires. First, as the percentage utilization of the FPGA increases, there are less paths available to connect two given nodes in the circuit. On average, this leads to longer paths and delays between nodes, which limits the maximum clock rate. Second, large operators such as dividers and square roots may have significant latencies. To increase the clock rate, they have to be optimized for speed or complemented with pipeline elements, which increases their resource usage.

The available resources on the TM-4 can support up to three One Iteration blocks, however this requires that each FPGA is used to its maximum capacity (98%), which limits the maximum speed of the system considerably. As mentioned previously, the sequence of One Iteration blocks can only process a new preliminary feature every several hundred clock cycles, and so it needs to run as fast as possible to avoid overflow in the MH buffer. For these reasons, the current implementation of the detector instantiates only two iterations, which are distributed among three FPGAs. This setup has the extra advantage of leaving some unused memory on FPGA 1 in

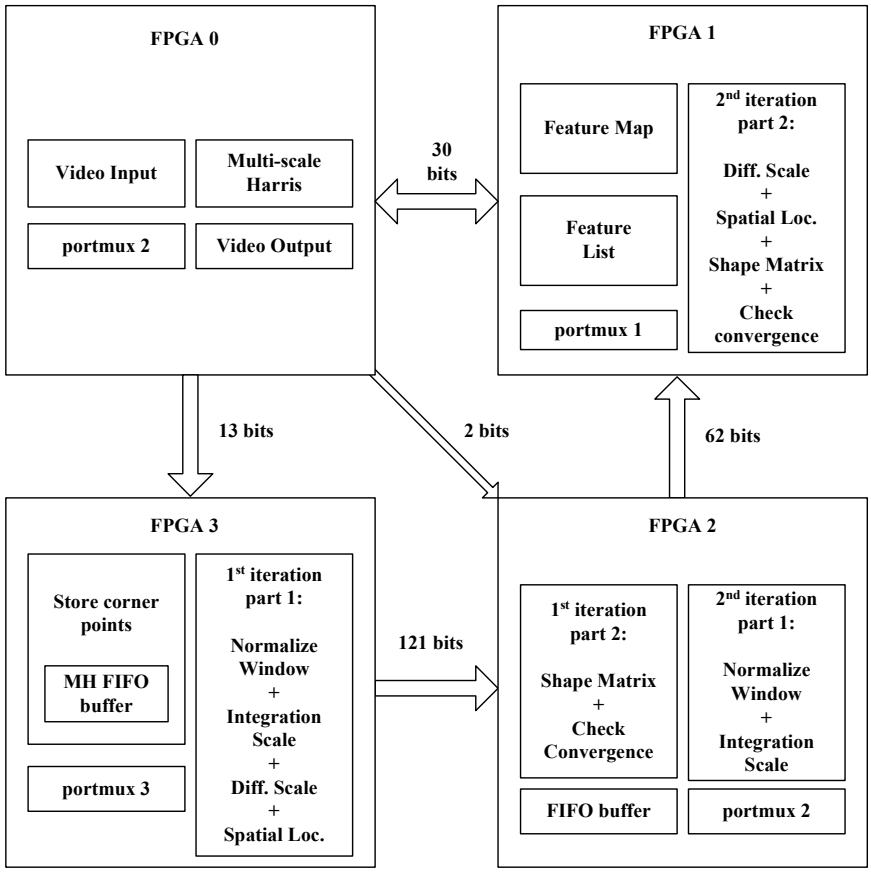


Figure 3.13: Distribution of the system over four FPGAs on the TM-4

case intermediate results need to be stored locally and exported through the TM-4 ports package for further processing.

Figure 3.13 shows the distribution of the system in the TM-4. All FPGAs include portmux components that are used to transmit parameters, control signals and results between the TM-4 and the terminal. In addition, a time-multiplexer circuit (Figure 3.14) is used to connect the modules in FPGA 2 and FPGA 1, since there are not enough inter-chip connections available to allow all data and control signals to be transmitted simultaneously.

The final results of the system are stored in a Feature Map and a Feature List. The Feature Map is a binary map that indicates the location of the features that are

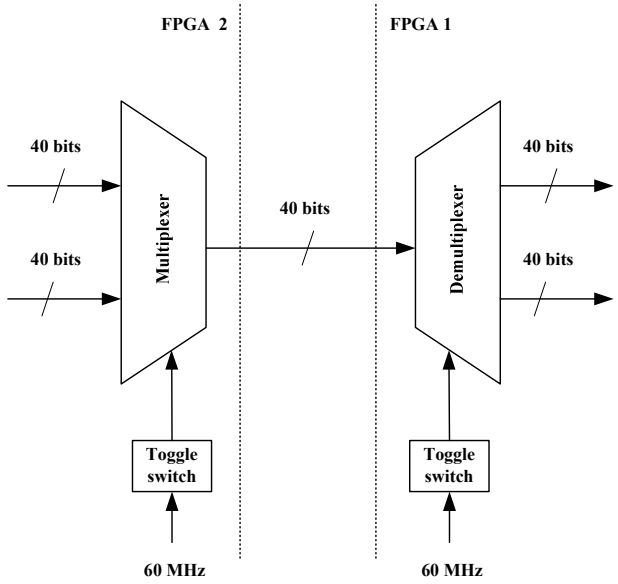


Figure 3.14: Time-multiplexer circuit

either accepted or in progress at the end of the second iteration. It is implemented as a double buffer circuit, where one buffer is updated with new locations while the other one is merged with the video stream to create the output to the VGA monitor.

The Feature List is implemented in on-chip RAM and contains the location ($\mathbf{x}$), integration scale (σ_I) and shape matrix (U) of the 8192 latest features produced by the system (accepted or in progress). It is connected to the portmux component on FPGA 1, so that the feature information can be read from the terminal.

3.6 Fixed-point representation analysis

Although a floating-point representation can achieve great precision and large dynamic ranges, it is often prohibitive to use floating-point operators in an FPGA implementation because of the large amount of hardware resources required to implement the normalization operations. This is specially true in hardware vision systems where operators are replicated numerous times and used in parallel to increase the speed of the system.

In vision algorithms, parameters and intermediate results rarely exploit the entire capacity of a floating-point representation. As a result, it is possible to study their precision and dynamic ranges to determine the optimal number of integer and fractional bits that need to be allocated to represent them in a fixed-point format. Larger bit-widths result in smaller quantization errors, however they require larger operators (such as dividers, multipliers, square roots and adders), more memory for buffering and higher inter-chip bandwidth.

In general, finding the optimal bit-widths for each variable is a complex multi-variable optimization problem that requires knowledge of both the algorithm and the target hardware platform. Among others, the following conditions have to be taken into account:

- Some variables have a larger impact on the final results than others, and so the most critical ones should have larger bit-widths.
- Different input images may yield different optimal widths.
- An efficient implementation must take into account the relative size of the operations associated with each variable and how many times these operations are repeated throughout the circuit, to try to minimize the width of the operations that require the most resources. Figure 3.15 shows the number of LUTs required to implement different operators as a function of the size of the input operands. These numbers correspond to operands with one clock cycle of pipeline latency and do not take into account that multipliers and adders can be implemented in embedded dedicated circuitry in some FPGA devices. As a reference, a Stratix S80 FPGA contains approximately 80000 LUTs.
- In FPGAs, memory blocks and dedicated arithmetic circuitry, such as embedded multipliers and adders, are optimized for signals of certain sizes (usually powers of 2). Although it is possible to use them with signals of other sizes, often the

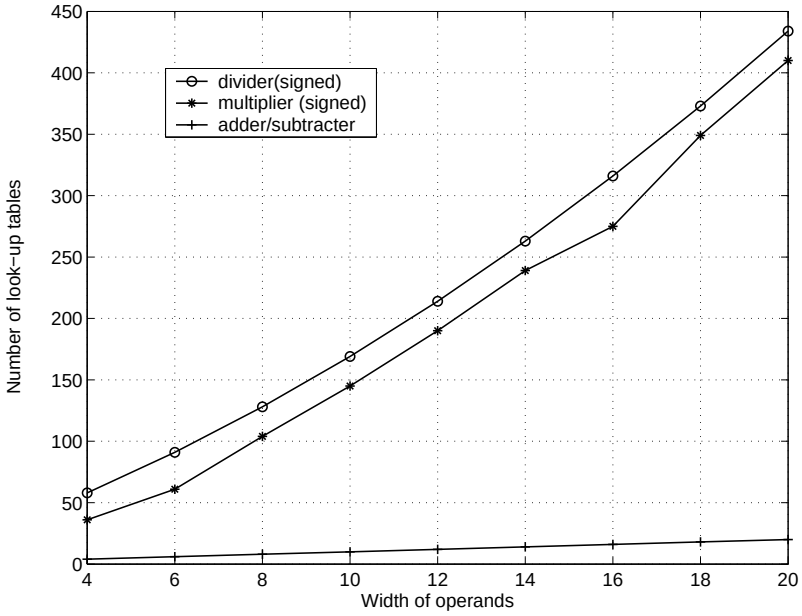


Figure 3.15: Number of look-up tables vs. size of operands for some common operators

(a) boat (b) graffiti (c) wadham

Figure 3.16: Test images

remaining resources cannot be used for other operations. To avoid waste of resources, signals should be kept in standard sizes whenever possible.

In this work, the choice of bit-widths was based on the results of floating-point and fixed-point MATLAB emulations on three images, shown in Figure 3.16. This approach does not yield the optimal bit widths for all quantities, but it makes the problem more manageable and gives the designer more flexibility to accommodate the conditions mentioned previously.

First, the floating-point version was used to determine the dynamic range of some

critical intermediate results, such as the *cornerness* values produced by the Multiscale Harris module and the values of the elements of the shape matrix U. This information facilitated choosing the number of integer bits allocated to some of the signals.

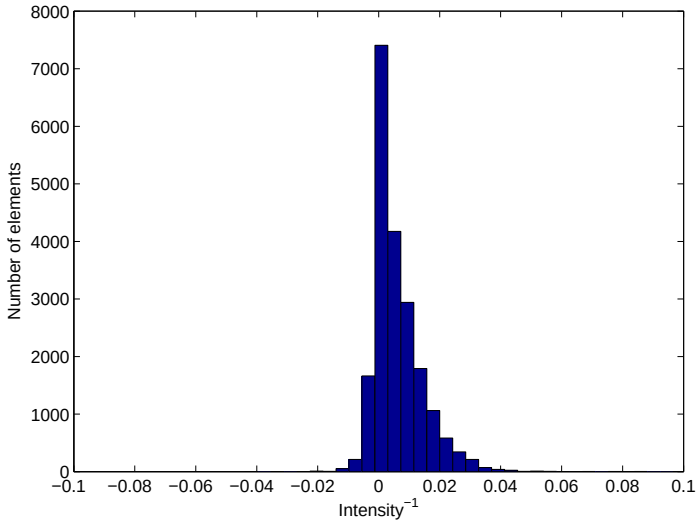
Figure 3.17 shows a sample distribution of the elements of the shape matrix before normalization for the three test images. Based on these results, most of the bits in the representation of U were allocated the fractional part.

Second, the fixed-point implementation was used to study the effect of varying the widths of both the integer and fractional parts of most parameters and some intermediate results. The widths were tested starting with the input parameters (σ_{N0}, ε, s) and continuing with intermediate results in the direction of information flow. Whenever possible, one width was varied while all others remained constant.

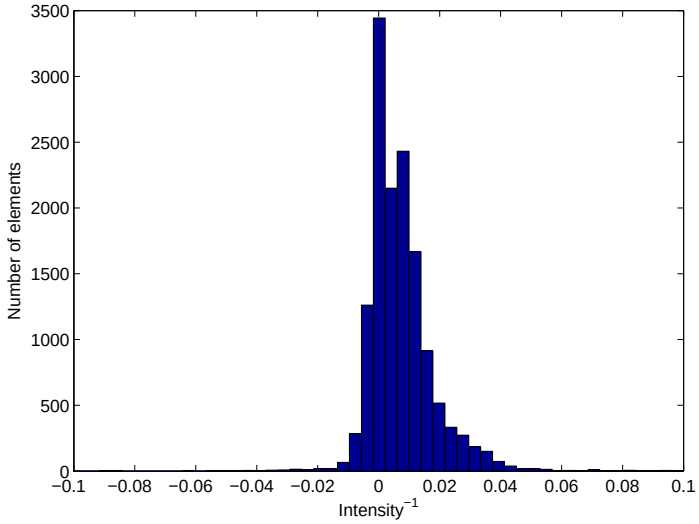
In order to compare the performance for different bit-widths, the location, integration scale and shape matrix of features in each of the test images were computed using the floating-point implementation and the fixed-point implementation with various widths of a specific variable. The results for each of the features produced by the fixed-point emulation were then compared to the results for the floating-point feature at the closest location.

Figure 3.18 shows the mean error in location, the total number of features detected and the median of the error in integration scale as a function of the width of the fractional part of the Gaussian coefficients used for filtering. For this variable, a width of 8 bits provides a reasonable tradeoff between size and performance. These results are a typical sample of the kinds of results obtained when only one width is varied at a time. As expected, the differences between the fixed-point and floating-point emulations approach a steady state as the number of bits increases, however they are not zero because of the constraints on the widths of other variables.

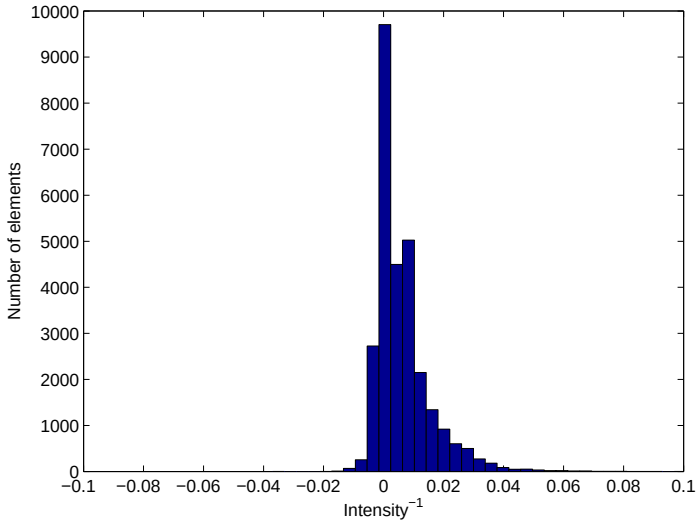
Another important consideration concerns the effect of restricting the Gaussian kernels used for image smoothing to discrete 11-tap filters. Figure 3.19 shows the Gaussian curves obtained for the scales $\sigma = 2$, $\sigma = 4$, $\sigma = 6$ and $\sigma = 8$. The 11-tap



(a) boat



(b) graffiti



(c) wadham

Figure 3.17: Distribution of the elements of the shape matrix U before normalization

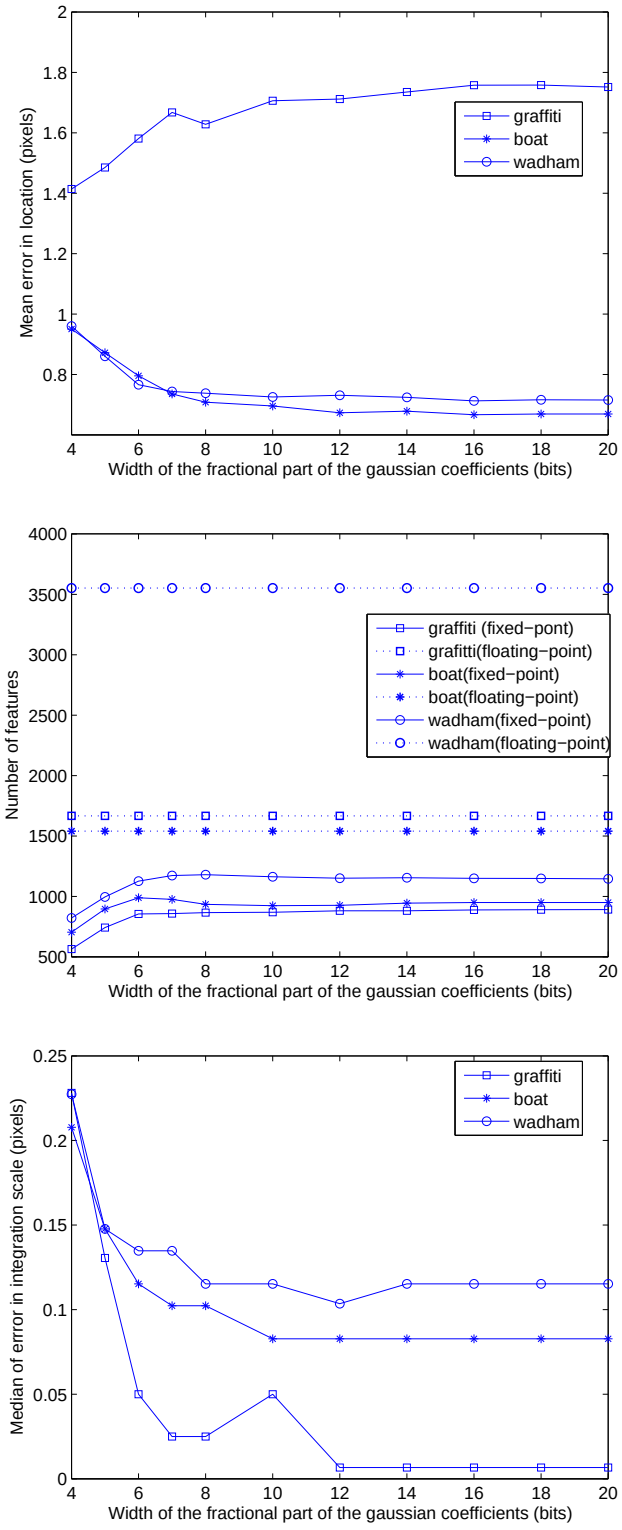


Figure 3.18: Detection errors for various bit-widths of the Gaussian coefficients. Mean of the error in location(top), number of features(center) and median of the error in scale (bottom)

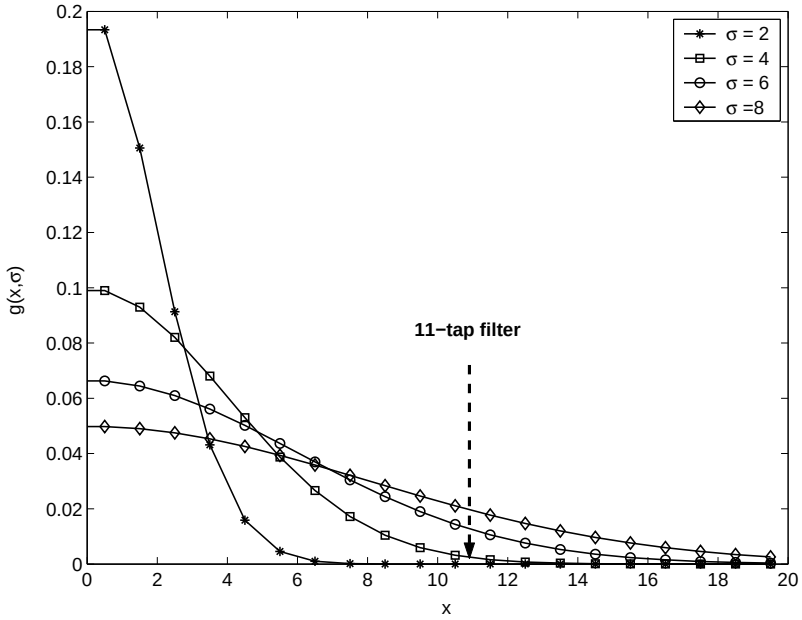


Figure 3.19: Effect of limiting the number of coefficients in Gaussian filters

filters in this implementation sample the curves up to the dotted line. The curves are properly sampled for $\sigma = 2$ and $\sigma = 4$, but the filter starts missing significant information for scales larger than $\sigma = 4$.

Table 3.1 shows the widths of the input parameters as set in the hardware implementation. These should be taken into account when passing parameters to the system to avoid accidental overflow or loss of precision.

Parameter Name	sign bit	integer bits	fractional bits
σ_{N0}	-	3	5
ε	-	1	4
ε^{-1}		1	4
s	-	-	4
α	-	1	4
ϵ_{conv}	-	1	3
ϵ_{div}	-	4	-
U_{ij}	1	3	6
s_1, s_2, s_3	-	-	8
Q threshold	-	-	4
Gaussian coefficients	-	-	8
cornerness threshold	-	32	-

Table 3.1: Number of bits allocated to the input parameters

Chapter 4

Results

This chapter presents a quantitative evaluation of the performance of the feature detector. Section 4.1 lists the hardware resources used in the system. Section 4.2 compares the computational speed of the system against the speed of software implementations. Finally, Section 4.3 compares the location, scale and shape of the features detected in hardware with the ground-truth values from the floating-point MATLAB implementation.

4.1 Hardware resources

Tables 4.1 and 4.2 list the amount and kinds of hardware resources used in the system. Table 4.1 lists the resources used in each of the modules described in Chapter 3, while Table 4.2 quotes the total resource utilization per FPGA. The numbers in parenthesis indicate the percentage of the total resources of that kind available in one Stratix S80 FPGA. The amounts marked *ext.* refer to external memory. The category *Others* groups glue logic, delays and I/O modules not included in the other categories. As reference, a Stratix S80 FPGA includes 79040 logic elements, 176 9-bit DSP elements and 7427520 bits of memory.

As seen in Table 4.2, the available resources in the Transmogrifier-4 FPGAs are

not fully utilized. This is due to the fact that the One Iteration module was originally designed to fit in a single Stratix S80 FPGA and it was later divided over two FPGAs to improve its speed. As a consequence, there are available unused resources that could be used to further improve the performance of the system.

Module	Logic elements	Memory bits	DSP elements
Video Input	1643 (2.07%)	1179648 (15.88%) 17694720 ext.	0
Video Output	1319 (1.66%)	1179729 (15.88%) 17694720 ext.	0
Multiscale Harris	34723 (43.93%)	1168012 (15.73%)	126 (72%)
MH FIFO buffer	455 (0.57%)	360448 (4.85%)	0
Normalize Window	3987 (5.04%)	3934666 (52.97%)	20 (11.36 %)
Integration Scale	22793 (28.84%)	17230 (0.23%)	5 (2.84 %)
Differentiation Scale	40116 (50.75%)	51754 (0.69%)	108 (61.36%)
Spatial Localization	4222 (0.53 %)	0	4 (2.27%)
Shape Matrix	9620 (12.17%)	402 (0%)	0
Intermediate FIFO buffers	208 (0.26 %)	145408 (1.96%)	0
Feature Map	353 (0.44%)	491520 (6.62%)	0
Feature List	209 (0.26%)	557056 (7.49%)	0
Portmux components	575 (0.72%)	0	0
Others	876 (1.11%)	4006261 (53.94%)	0

Table 4.1: Resource utilization per module

4.2 Speed

One of the most important measures in the evaluation of a hardware implementation is how it compares to a software implementation in terms of speed.

Table 4.3 shows the processing times for the hardware system and the floating-point MATLAB implementation for the case of a single image. The times are quoted along with the corresponding value of the *cornerness threshold* parameter used to

FPGA	Logic elements	Memory bits	RAM block bits[1]	DSP elements
FPGA 0	38256 (48%)	3527389(47%)	6708672 (90%)	126 (72 %)
FPGA 1	55028 (70 %)	5045677 (68%)	6009408 (81%)	112 (64%)
FPGA 2	37036 (47 %)	4159329 (56%)	5474304 (74%)	25 (14%)
FPGA 3	70159179 (89 %)	4437738 (60%)	5586624 (75%)	117 (66%)

Table 4.2: Resource utilization per FPGA

obtain them, and the number of preliminary features detected in the Multiscale Harris stage for these thresholds.

The processing times for the floating-point implementation and the number of preliminary features were measured directly from MATLAB running on a 2.66 GHz Pentium IV processor with 4 GB of memory. The processing times for the hardware system were computed for clock rates of 12 MHz (for the Multiscale Harris module) and 60 MHz (for the iteration blocks) using the formula

$$hardware\ time = system\ latency + \left(\begin{array}{c} \#\ of\ preliminary \\ features \end{array} - 1 \right) \times pipeline\ latency \tag{4.1}$$

where

$$system\ latency = MH\ latency + One\ Iteration\ latency \times \#\ of\ iterations \tag{4.2}$$

The *system latency* is the (fixed) time it takes to detect and refine a single feature, from the input to the Multiscale Harris module (MH) to the output of the last iteration block. The times in Table 4.3 were computed for $\#\ of\ iterations = 2$. Because the system is almost entirely pipelined, several feature points can be processed at the various stages of the system simultaneously. The *pipeline latency* is the delay between the times in which consecutive preliminary features are retrieved from the

[1]Quartus II spreads logical memories into multiple RAM blocks to improve timing. The term 'RAM block bits' is the number of blocks used times the number of bits available in each block. The term 'Memory bits', on the other hand, refers to the actual number of RAM bits used, regardless of their locations.

Cornerness threshold	# of preliminary features	Time (s)		Ratio of floating-point time to hardware time
		floating-point	hardware	
500	11307	1.1057×10^3	0.1232	8.8765×10^3
1000	10768	1.0394×10^3	0.1173	8.8583×10^3
2000	9828	0.9558×10^3	0.1071	8.9200×10^3
3000	9016	0.8824×10^3	0.0983	8.9715×10^3
4000	9371	0.8267×10^3	0.0914	9.0480×10^3
5000	7821	0.7782×10^3	0.0854	9.1114×10^3
6000	7332	0.7373×10^3	0.0801	9.2034×10^3
7000	6905	0.7056×10^3	0.0755	9.3469×10^3
8000	6577	0.6738×10^3	0.0719	9.3666×10^3
9000	6280	0.6461×10^3	0.0687	9.4015×10^3
10000	6009	0.6230×10^3	0.0658	9.4705×10^3
12000	5588	0.5876×10^3	0.0612	9.5970×10^3
14000	5230	0.5562×10^3	0.0573	9.6994×10^3

Table 4.3: Processing times for the hardware and MATLAB implementations for 2 iterations

MH FIFO buffer. Since the number of clock cycles corresponding to the system and pipeline latencies are known from the design of the circuit, the values of the latencies in seconds can be calculated by multiplying the number of cycles by the corresponding clock rate. After an initial latency of *system latency* $= 6.9725 \times 10^{-5}$ seconds, the system outputs the location, scale and shape of a new feature every *pipeline latency* $= 1.0833 \times 10^{-5}$ seconds. Hence, the total processing time is directly related to the number of preliminary features detected.

For comparison purposes, the calculation of the processing times for the hardware system assumes the following:

- The number of preliminary features detected in the software and hardware implementations is the same.
- No features are dropped from the MH FIFO buffer.
- All preliminary features are processed by both iterations (this applies for both software and hardware). This is a worst-case scenario, since in general a portion

of the features will be discarded after the first iteration.

Table 4.3 shows that the hardware implementation achieves a speed that is on average 9.2×10^3 times faster than the speed of the MATLAB version. In general, MATLAB implementations are not the most efficient in terms of speed, however the majority of operations involved in the feature detector algorithm are matrix operations, which are highly optimized in MATLAB. For this reason, it is expected that software implementations in other languages, like C, would improve the performance of the MATLAB version by up to a factor of 100. Taking this improvement into account, the hardware implementation achieves an increase in speed of at least two orders of magnitude.

A sample processing time for a software implementation by Mikolajczyk is provided in [37]. In this implementation, 1123 points are detected in a 800×640 image, for 5 integration scales and a variable number of iterations (typically less than 10). Recursive filters are used to accelerate the Gaussian filtering. The detection is performed in 36 seconds on a 500 MHz Pentium II processor. Although there are not enough details available to draw a precise comparison between this software implementation and the hardware system, a rough estimate is that the hardware implementation improves the speed of the software version by a factor of 100 for the same number of detected points.

4.3 Accuracy

The fixed-point representation and other algorithmic simplifications used in the hardware implementation of the feature detector introduce errors in the location, scale and shape of the detected features. To provide a quantitative measure of these errors, the results obtained from ModelSim simulations of two test images, shown in Figure 4.1, were compared against the results obtained from processing the same images with the floating-point MATLAB implementation. Each image was analyzed at three scales:

(a) graffiti (b) cars

Figure 4.1: Test images

$\sigma_{N0} = 1$ pixel, $\sigma_{N0} = 2$ pixels and $\sigma_{N0} = 4$ pixels. The values assigned to the other input parameters are listed in Table 4.4. For all results, the parameters used in the simulations were the same as the ones used in the floating-point MATLAB implementation.

Table 4.5 shows the number of features detected in each test case by the hardware and MATLAB implementations. The preliminary features found by the Multiscale Harris module are shown in the *initial* column and the number of features remaining after two iterations are quoted in the *final* column. The final features include those that have converged, marked as accepted (*acc*) and those that are still in progress (*ip*). In general, given a set of fixed input parameters, more features are detected in software and at the smaller scales. This is explained by the fact the algorithm is largely based in choosing maxima or minima of intermediate results in local neighbourhoods. In the case of the software implementation, the larger precision provided by the floating-point representation allows the detector to differentiate between variables of similar value, which may appear to the hardware implementation as identical because of truncation in the fixed-point representation. Similarly, larger integration scales, 'smooth out' details in the image and other signals, resulting in a smaller number of local extrema.

Figures 4.2 to 4.9 show the results of comparing the location $\mathbf{x}_{new}$, integration

Parameter	Graffiti Image	Cars Image
Number of iterations	2	2
Gaussian Table: number of scales, range (pixels), step (pixels)	60, [0.5 – 7.845], 0.125	60, [0.5 – 7.845], 0.125
σ_{N0} (pixels)	1, 2, 4	1, 2, 4
ε	1.125	1.125
s	0.6875	0.6875
α ($\frac{\text{intensity}^2}{\text{pixel}^2}$)	0.125	0.125
s_1	0.5	0.5
s_2	0.5977	0.5977
s_3	0.6992	0.6992
Cornerness threshold (intensity^4)	50	50
LoG threshold (intensity)	0	0
Q threshold	0	0
ϵ_{conv} ($\frac{\lambda_{min}(\mu)}{\lambda_{min}(\mu)} \times 100\%$)	1.1250 (89 %)	1.1250 (89 %)
ϵ_{div}	15	15

Table 4.4: Values of the parameters used to compute the results in Section 4.3.

scale σ_{Inew} and shape adaptation matrix U computed by the hardware and MATLAB implementations in each test case. Features in the two implementations were matched by their spatial location. In the cases where there was more than one matching feature, the scale and shape matrix were used to choose among them. In addition, features within 20 pixels of the image border were discarded to avoid edge effects. Two features were considered to *correspond* if their spatial location did not differ by more than 3 pixels. The results for the scale and shape adaptation matrix shown in Figures 4.4 to 4.9 were computed from the pairs of matching features that satisfied this distance threshold. It is worth noting that the results are likely to vary if the number of iterations is increased, since only a fraction of the total remaining features converge by the second iteration, as shown in Table 4.5.

Test case		MATLAB		Hardware	
Image	σ_{N0}	Initial	Final (acc/ip)	Initial	Final (acc/ip)
Graffiti	1	1917	1104 (163/941)	2017	488 (24/464)
	2	1038	597 (77/520)	1234	397 (32/345)
	4	749	473 (50/423)	1040	380 (18/362)
Cars	1	4721	3324 (438/2886)	4769	723 (55/668)
	2	1596	1109 (111/998)	1955	810 (55/755)
	4	584	376 (40/336)	811	257 (10/247)

Table 4.5: Number of features after the Multiscale Harris module (*initial*) and after two iterations (*final*), for each test case. The final features consist of those that have converged (*acc*) and those that are still in progress (*ip*).

Figures 4.2 and 4.3 show the distribution of Euclidian distances between the features detected in hardware and the MATLAB features at the closest spatial location. The majority of hardware features are located within 3 pixels of a software feature. The tails of the histograms correspond to hardware features that did not have a match among the software features. The peaks of the histograms are around 1 pixel, which can be attributed to the limited precision imposed by the fixed-point representation. The update equations for $\mathbf{x}_w$ and $\mathbf{x}_{new}$ (equations 3.10 and 3.16) produce non-integer coordinates that are fully utilized in the software implementation. On the other hand, the hardware implementation only uses 2 bits to encode the fractional parts of $\mathbf{x}_w$ and $\mathbf{x}_{new}$, and therefore any changes in the coordinates are truncated to steps of 0.25 pixels. Moreover, the fractional bits are dropped at the output of the detector to reduce the amount of on-chip memory required to store the results in the Feature List buffer, which is not necessary in the software implementation.

Figures 4.4 and 4.5 present the distribution of the absolute value of the difference between the scales of corresponding features, for each test case. The histograms show that the errors in scale are small, in particular when compared to the the step between scales of $\varepsilon = 1.125$, and can be reasonably attributed to quantization errors in the

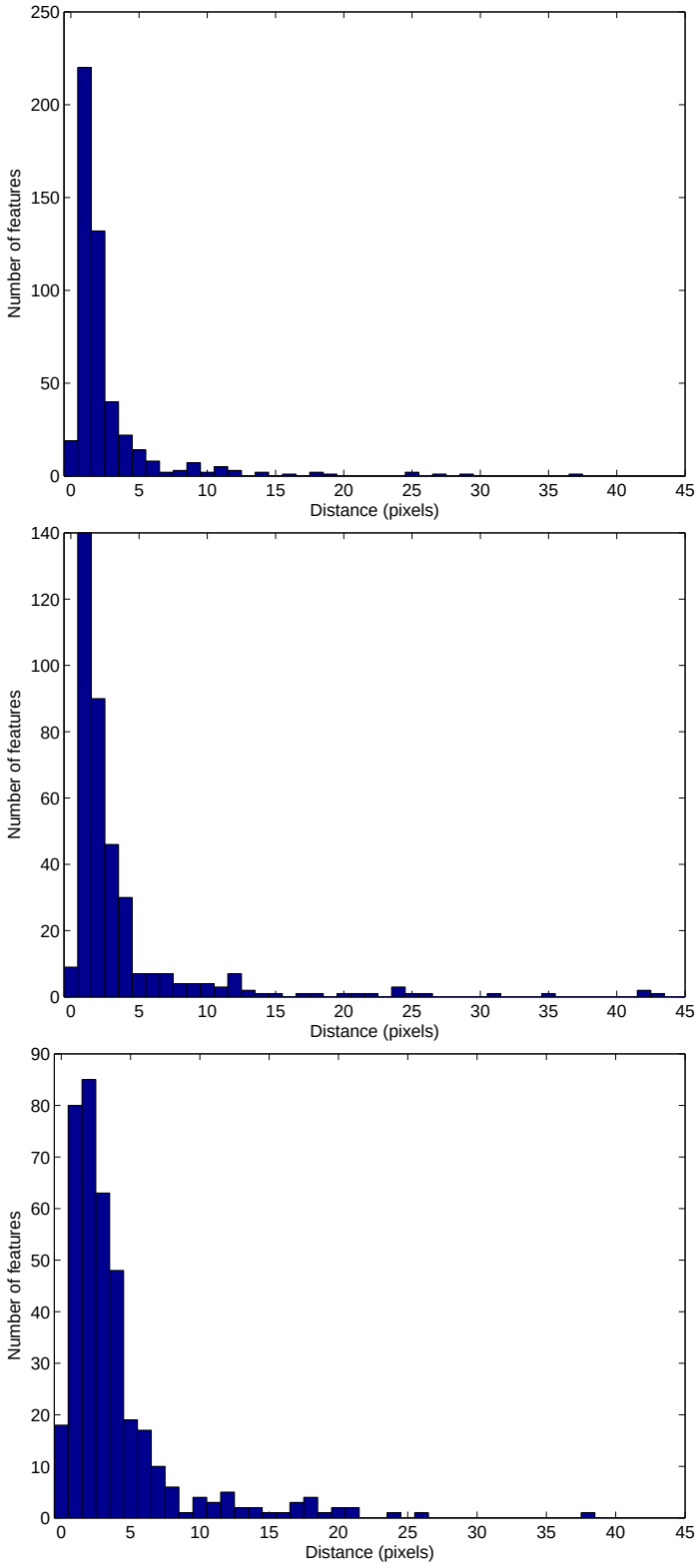


Figure 4.2: Distribution of Euclidian distances between matching hardware and software features in the Graffiti image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

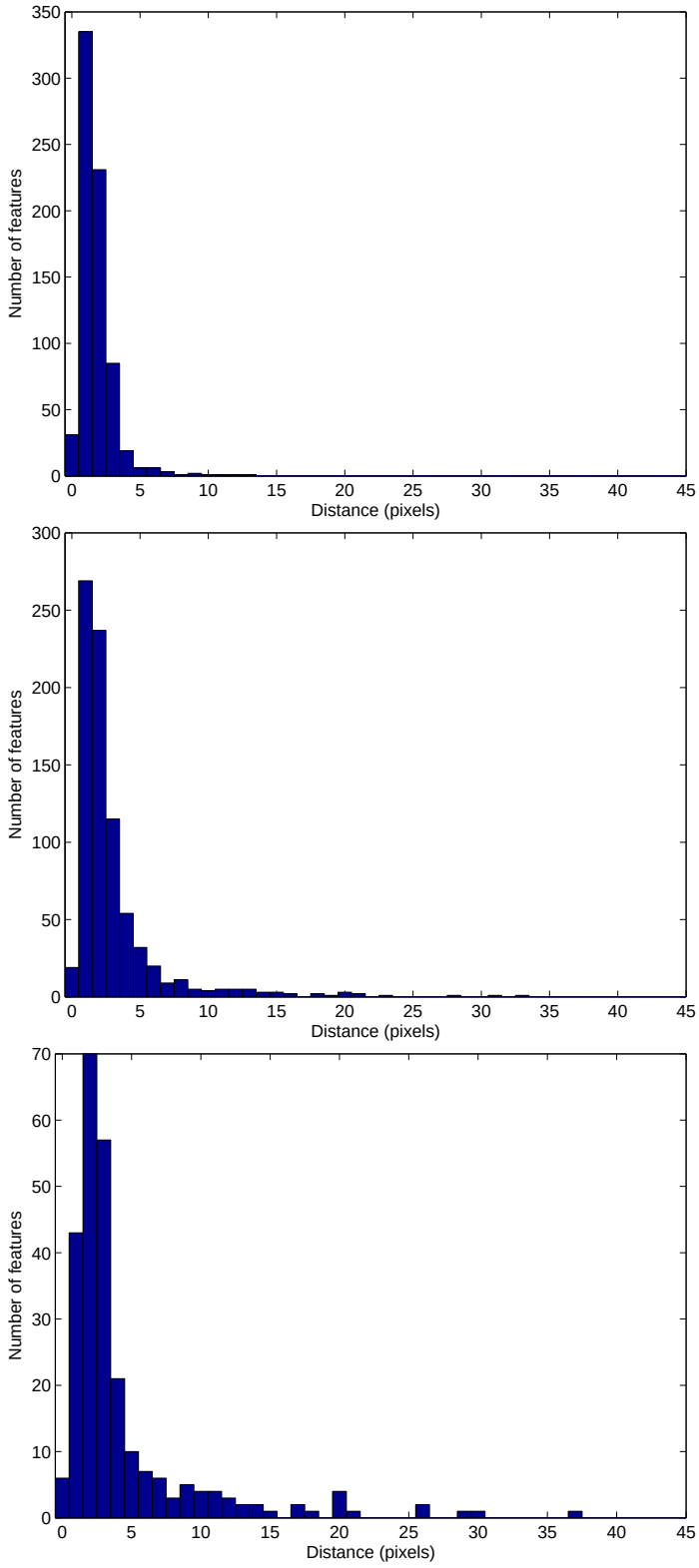


Figure 4.3: Distribution of Euclidian distances between matching hardware and software features in the Cars image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

hardware implementation. This shows that the scale of an image structure can be estimated reliably despite of small errors in the location of the feature.

As discussed in Chapter 2, the shape adaptation matrix U represents an affine transformation that warps the image around a feature so that the local second moment matrix in the transformed image has equal eigenvalues. The transformation can be described in terms of the magnitude and direction of the non-uniform scaling it produces. This idea is used to compare the shape matrices U_H and U_S, obtained with the hardware and software implementations respectively. The first error measure is the ratio R of the *stretch* produced by the U matrices of corresponding features,

$$R = \frac{stretch(U_H)}{stretch(U_S)}, \tag{4.3}$$

where

$$stretch(U_H) = \frac{\lambda_{min}(U_H)}{\lambda_{max}(U_H)} \tag{4.4}$$

and

$$stretch(U_S) = \frac{\lambda_{min}(U_S)}{\lambda_{max}(U_S)}. \tag{4.5}$$

The values $\lambda_{min}(U_H)$, $\lambda_{max}(U_H)$, $\lambda_{min}(U_S)$ and $\lambda_{max}(U_S)$ are the minimum and maximum eigenvalues of U_H and U_S. It is worth noting that since the shape matrices are normalized to have a maximum eigenvalue of 1, the ratio R is, in theory, equivalent to the ratio of $\lambda_{min}(U_H)$ to $\lambda_{min}(U_S)$. In practice, however, the *stretch* functions have to be computed to take into account possible round-off errors in the normalization of U_H. Figures 4.6 and 4.7 show the distribution of the R ratios for each test case. The ratios have a slight tendency to be less than 1, which suggests that the shape adaptation matrix may be less developed in the hardware implementation than in the software implementation.

The directions of scaling produced by U_H and U_S were compared by measuring the angle between the eigenvectors $\mathbf{V}_H$ and $\mathbf{V}_S$ that correspond to the maximum eigenvalues of U_H and U_S, respectively. The eigenvectors were calculated using the

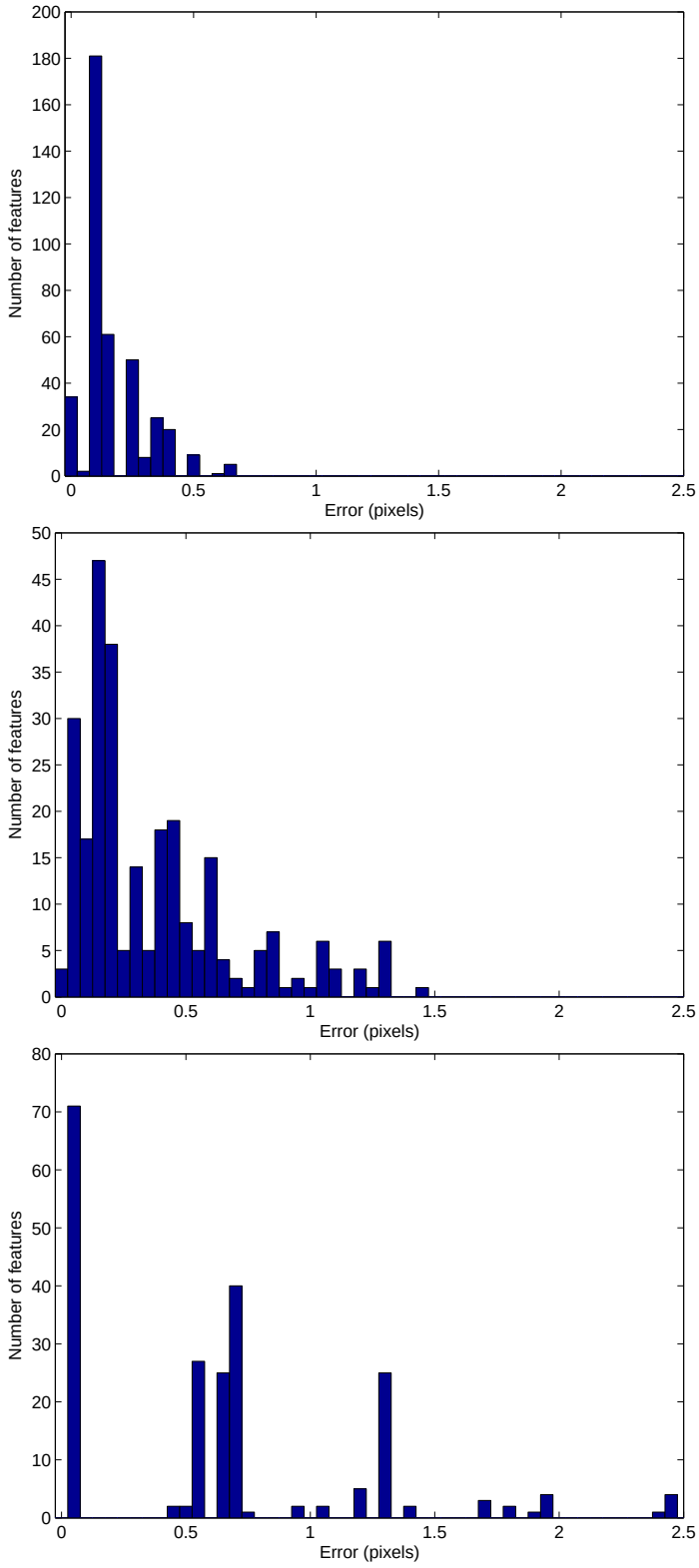


Figure 4.4: Distribution of errors in scale between corresponding hardware and software features in the Graffiti image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

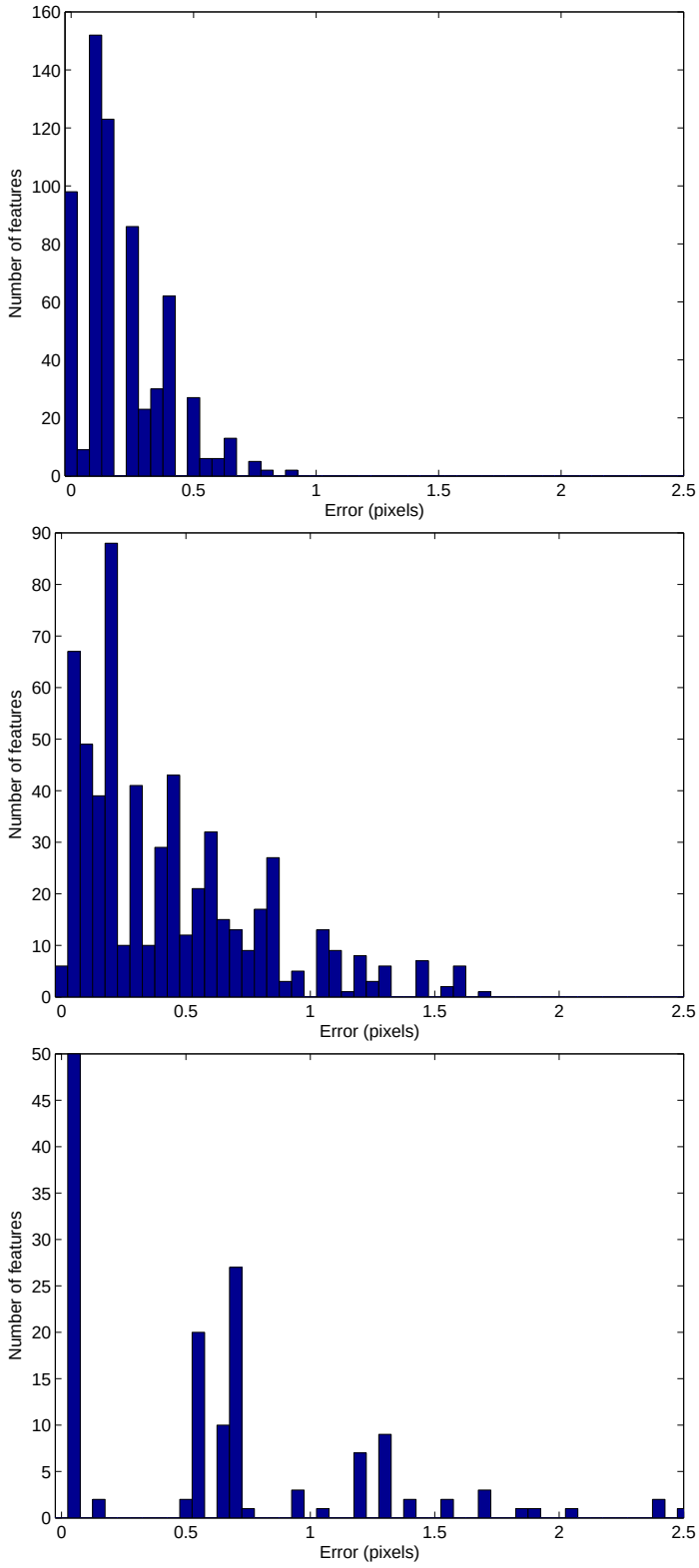


Figure 4.5: Distribution of errors in scale between corresponding hardware and software features in the Cars image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

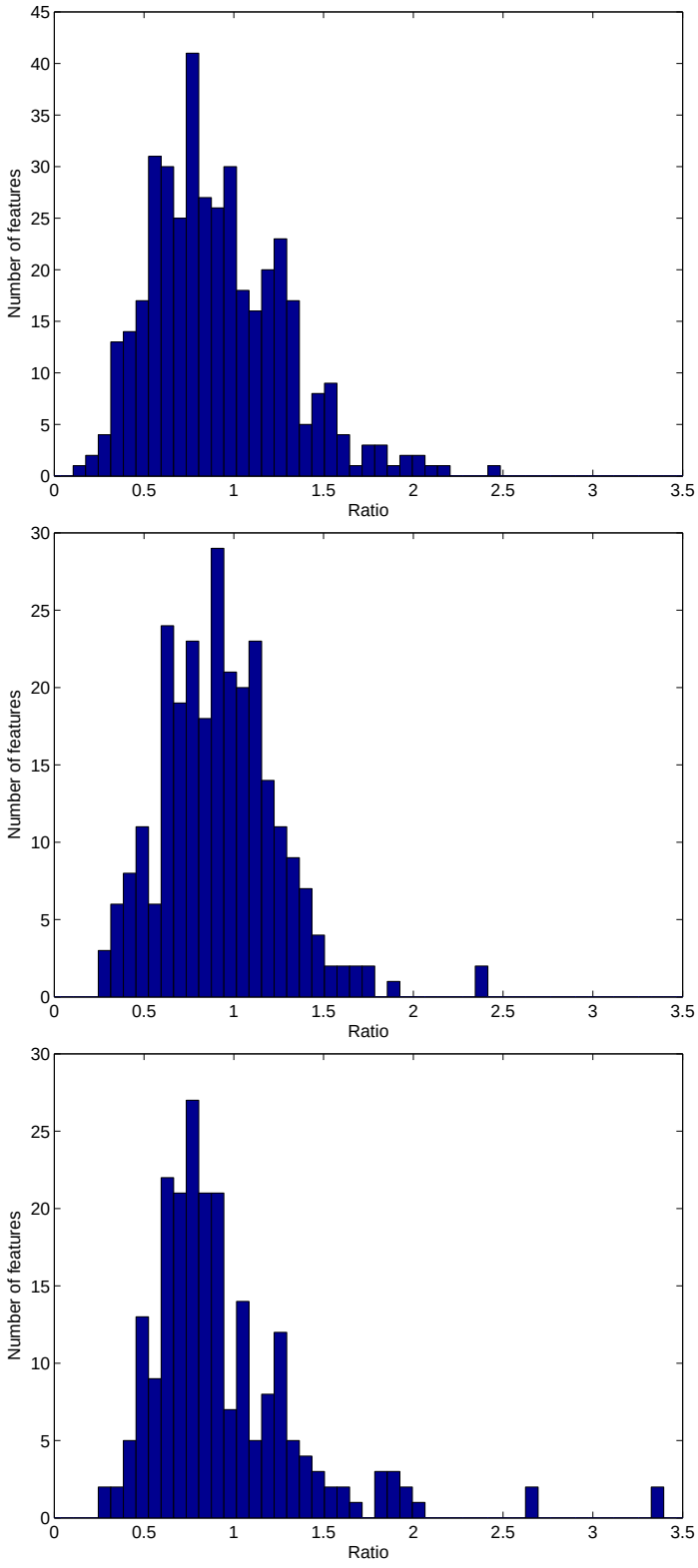


Figure 4.6: Distribution of the ratio $R = \frac{stretch(U_H)}{stretch(U_S)}$ in the Graffiti image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

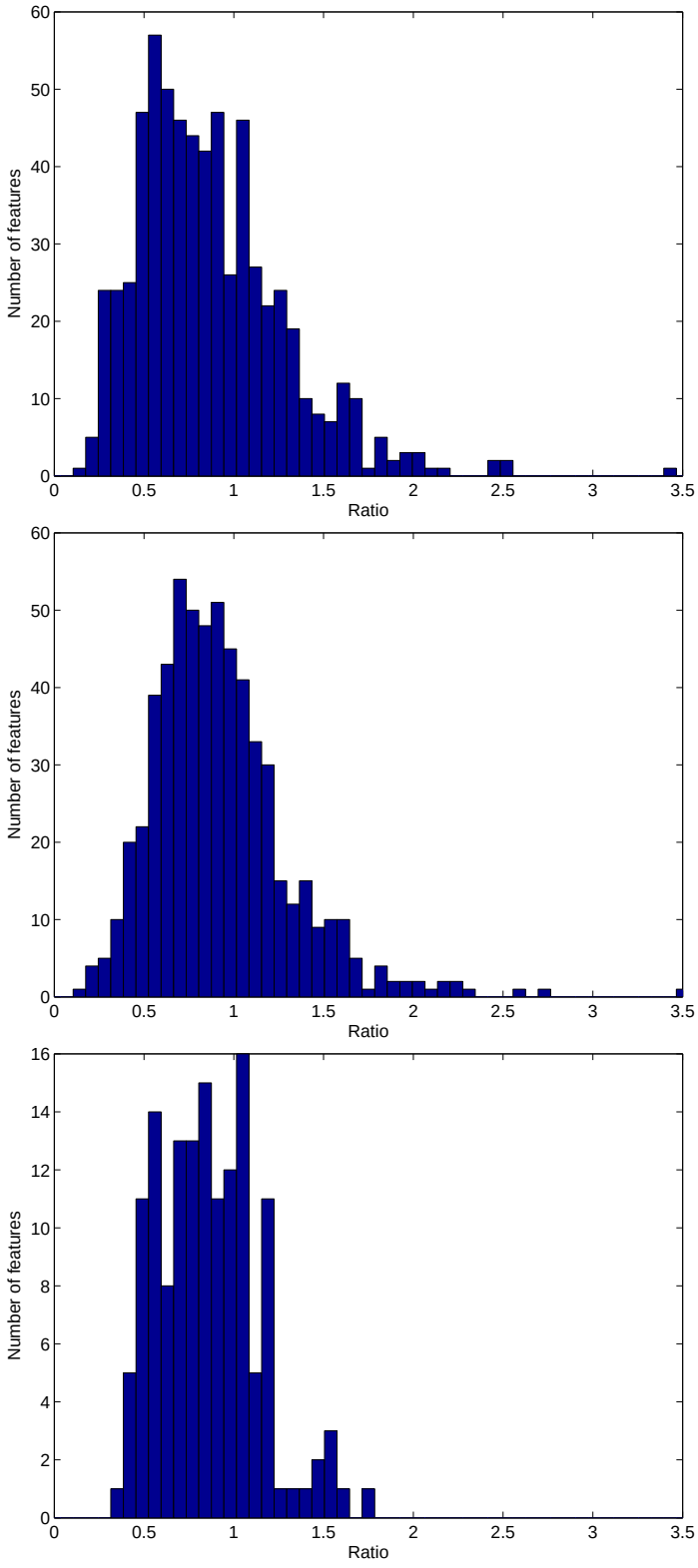


Figure 4.7: Distribution of the ratio $R = \frac{stretch(U_H)}{stretch(U_S)}$ in the Cars image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

MATLAB function *eig*, and the angle was computed as

$$angle = \min\left(\text{acosd}\left(\frac{\mathbf{V}_H \cdot \mathbf{V}_S}{||\mathbf{V}_H||||\mathbf{V}_S||}\right), 180^\circ - \text{acosd}\left(\frac{\mathbf{V}_H \cdot \mathbf{V}_S}{||\mathbf{V}_H||||\mathbf{V}_S||}\right)\right). \tag{4.6}$$

where *acosd(x)* is the MATLAB function that returns the inverse cosine of the argument x in degrees. The distribution of angles is shown in Figures 4.8 and 4.9.

From the distribution of errors for the location, scale and shape, it can be observed that the shape adaptation matrix is the most sensitive to quantization errors introduced by the fixed-point representation. As mentioned previously, most of the features used to obtain these results did not converge by the second iteration, and therefore it is possible that the errors in U_H are partly due to the different rates of convergence in the hardware and software implementations. That is, even in the case where the location and scale of corresponding features are the same, it is possible that their shape adaptation matrices are at different stages of convergence.

Figure 4.10 shows the percentage of accepted features (i.e., features that have converged) as a function of the number of iterations, for each test image. The percentages are calculated with respect to the total number of features remaining in the system, which consist of the accepted features and the features that are still in progress,

$$\%\ accepted\ features = \frac{\#\ \text{accepted}}{\#\ \text{accepted} + \#\ \text{in progress}} \times 100. \tag{4.7}$$

The results labelled *'fixed-point'* were computed using the fixed-point MATLAB implementation of the algorithm, since it takes a long time to simulate 20 iterations of the hardware circuit in ModelSim. The values of the parameters used are the same as those shown in Table 4.4 (with the exception of the number of iterations). The percentages show that the features computed with the fixed-point implementation converge more slowly than the features computed with the floating-point version. Moreover, some of the features do not converge in the fixed-point implementation, which leads to believe that they fall into local optima that do not satisfy the convergence criterion, due to lack of precision in either the location or the shape adaptation matrix.

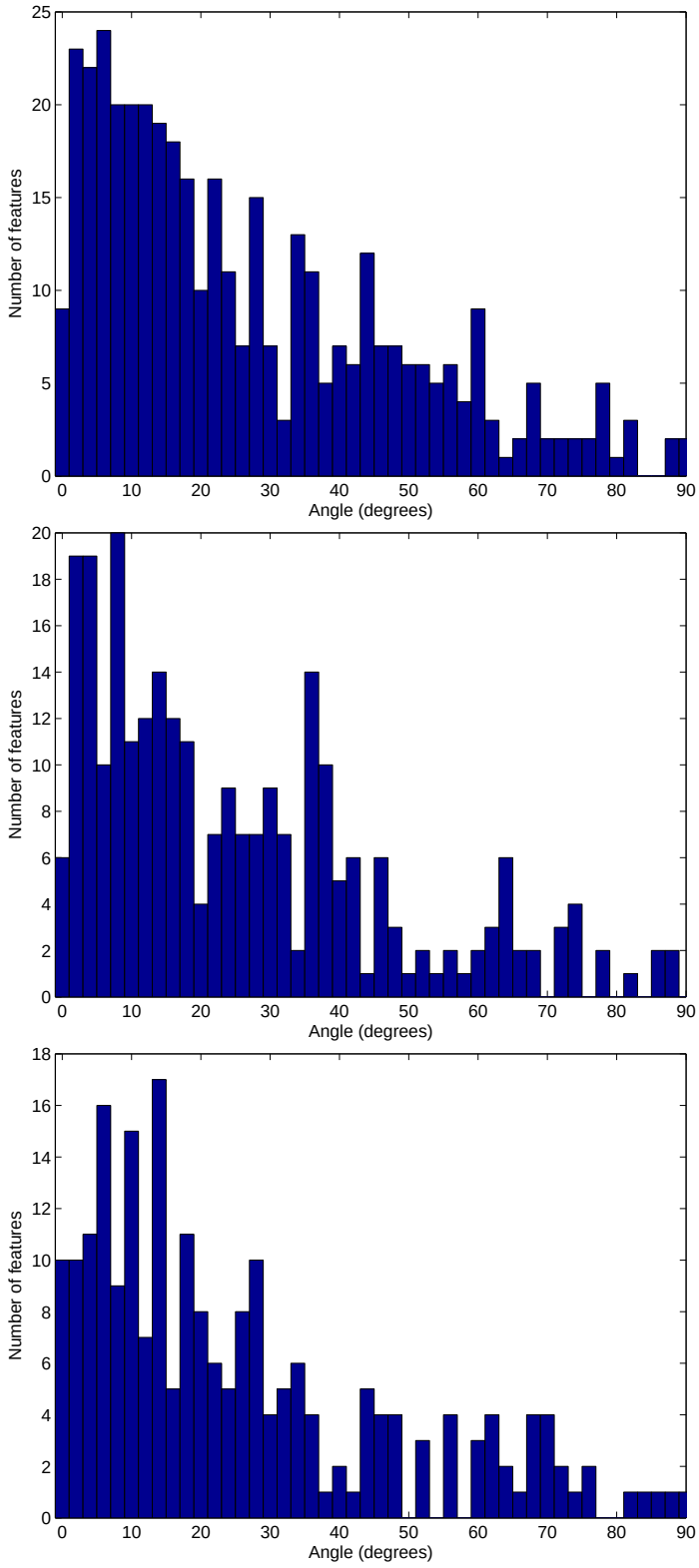


Figure 4.8: Distribution of angles between the eigenvectors $\mathbf{V}_H$ and $\mathbf{V}_S$ in the Graffiti image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

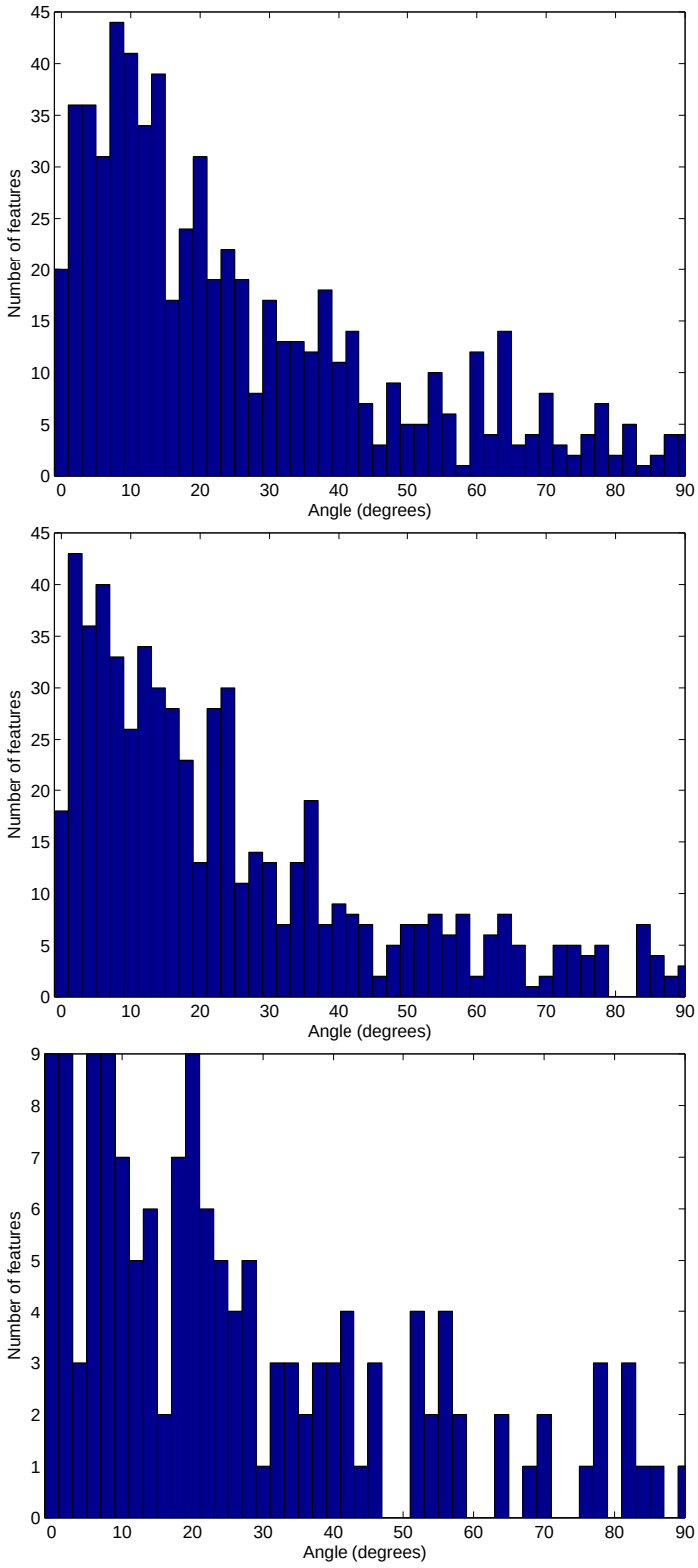


Figure 4.9: Distribution of angles between the eigenvectors $\mathbf{V}_H$ and $\mathbf{V}_S$ in the Cars image, for $\sigma_{N0} = 1$ pixel (top), $\sigma_{N0} = 2$ pixels (center) and $\sigma_{N0} = 4$ pixels (bottom)

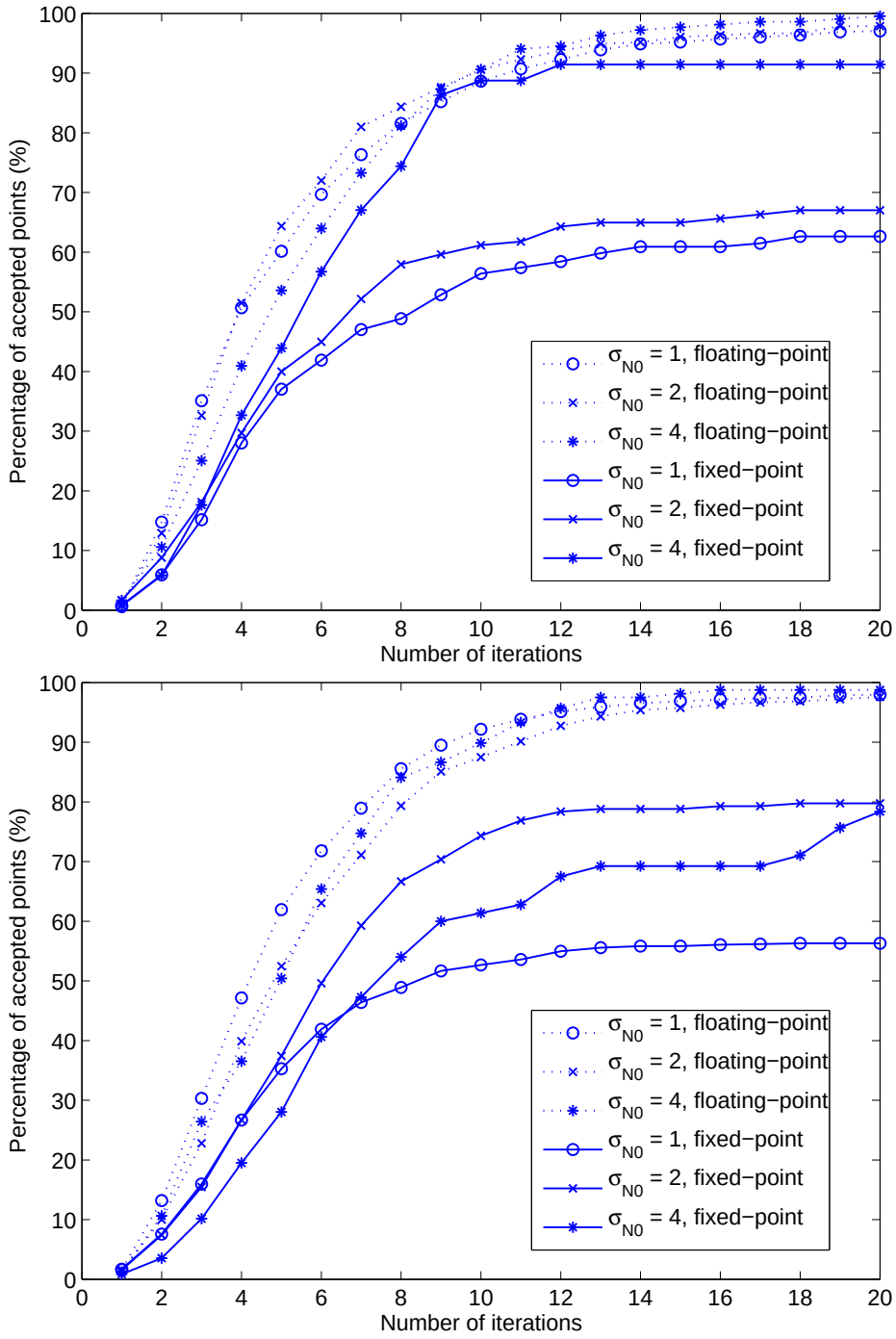


Figure 4.10: Percentage of accepted features as a function of the number of iterations for the floating-point and fixed-point MATLAB implementations. Top: Graffiti image. Bottom: Cars image.

4.4 Feature detector on the Transmogrifier-4

Figure 4.11 shows a sample output of the circuit running on the Transmogrifier-4. An image frame from the NTSC camera was saved in on-chip memory and then read from a UNIX terminal using the TM-4 *ports* package. Similarly, the location, scale and shape matrix of the features detected in the image were read from the Feature List buffer. The image and feature information were then imported to MATLAB, where the locations of the feature were superimposed on the image.

The information obtained from the TM-4 shows that there are some discrepancies between the ModelSim simulations and the live circuit. The location, scale and shape matrix of most of the features in the circuit are correct, however there are anomalies that can only be explained by failed timing requirements. This problem is often encountered in hardware designs since the simulations may not take into account all propagation delays between nodes in a circuit, in particular those concerning inter-chip connections. The circuit is currently being tested using the SignalTapII tool, which samples specified signals in the circuit, saves the samples in on-chip memory and allows the user to visualize the signals in time. The description of the circuit has to be recompiled every time probes are added or changed in the circuit, which increases the verification time. Taking this into consideration, it is expected that the circuit will be corrected and verified in 2–4 weeks.

4.5 Summary

The FPGA-based implementation of the Harris-Affine feature detector on the Transmogrifier-4 achieves a processing speed of 90–9000 times the speed of an equivalent software system, depending on the language of implementation and the computing platform. This increase in speed is possible because of the inherent parallelism of many of the operations in the detection algorithm, which allows the hardware implementation to perform numerous operations simultaneously.

Figure 4.11: Sample output of the feature detector running on the Transmogrifier-4. The original image is shown at the top. The location of the features obtained from the circuit have been superimposed on the original image and marked with white '+' signs

The location, scale and shape adaptation matrix of the features detected in the hardware implementation were compared to those obtained in a floating-point MATLAB implementation. The comparisons show that quantization errors caused by the fixed-point representation affect the accuracy of the hardware system, in particular that of the shape adaptation matrix. Nonetheless, the results are fairly consistent across images and scales, and therefore it may be possible to use the system to detect features that can be reliably matched across images. Moreover, since the current implementation does not fully utilize the available hardware resources on the Transmogrifier-4, it is possible to increase the number of bits used to represent sensitive signals to improve the accuracy of the system.

Chapter 5

Conclusions and Future work

This thesis presents an FPGA-based implementation of the Harris-Affine feature detector presented in [37]. The feature detector is coupled with NTSC and VGA video interfaces to create a smart camera system that provides the location, scale and shape of corner points detected in video frames at a rate of 30 frames per second.

The Harris-Affine detector produces features that are robust to image rotation and translation, as well as to significant changes in illumination, scale and affine deformations. This performance, however, comes at the expense of long processing times. The fact that many of the operations in the algorithm can be performed in parallel make it an ideal candidate for implementation in hardware, where the parallelism can be exploited to increase the processing speed.

Field-programmable gate arrays (FPGAs) offer a good hardware development platform because they can be easily reprogrammed to fit the requirements of a given application. This is particularly useful in the early stages of implementation of a new design since corrections and additional features can be added in short periods of time.

The main challenge with implementing a complex algorithm in hardware is to balance the need for numerical precision with an efficient use of the available hardware resources. This requirement is of special importance in an iterative algorithm such as the Harris-Affine detector, in which the accuracy of intermediate results can have

a significant effect on the way the algorithm converges. This implementation of the feature detector uses a fixed-point numerical representation, as opposed to the floating-point representation used in general-purpose processors, because of the large amount of resources required to implement floating-point operators.

Another important consideration in the design of the hardware implementation is that the amount of data that needs to be processed by the iterative portion of the detector can vary significantly depending on the nature of the images being processed and the value of the input parameters. This makes the design of a pipelined architecture more challenging, because of the unknown delay between consecutive features.

The results obtained from comparing the hardware implementation and the floating-point MATLAB model show that the location, scale and shape adaptation matrices computed in the hardware implementation are significantly affected by the fixed-point representation. From these, the shape adaptation matrix is the most affected. This is due to the fact that many operations are required to compute the matrix, resulting in accumulated quantization errors. Future improvements of this implementation should attempt to provide more bits to improve the precision of these operations, in particular the division and square root operations, which are currently implemented using integer operators. It is worth noting that a comparison of the convergence rates of the floating-point and fixed-point MATLAB implementations shows that the fixed-point precision slows the convergence of the features. Therefore, it is expected that the pairs of corresponding features used to compute the results are at different stages of refinement, which is likely to have an effect on the results.

The system described in this thesis is, to the knowledge of the author, the first hardware-based implementation of the Harris-Affine feature detector. As such, there is room for improvements and refinements, both in the accuracy of the results and in the processing speed. The following are a few ideas to guide any future work on this system:

- Given the total resource utilization for the current implementation it is possible to study how the resources that are still available could be distributed to improve the performance of the system. It is worth noting that the One Iteration module was originally designed to fit in a single Stratix S80 FPGA, and for that reason the number of bits allocated to the signals were tightly constrained by the available resources. This had a significant impact on the accuracy of the results, in particular that of the shape adaptation matrix.

  In addition, it is possible to use the available resources to optimize some of the larger operations (such as divisions) so that they can be clocked at a faster rate, which would have a direct effect on how many preliminary features can be processed by the iteration modules.

- The current 11-tap filters limit the scale of the Gaussian kernel that can be used for smoothing. A way to increase the reach of the filters is to perform consecutive convolutions to combine their effects. This, however, may require restricting the values of σ_{N0} and ε, which are currently only limited by the number of scales in the Gaussian look-up table.

- The One Iteration module could be used in a 'single-iteration' architecture where a single module implements all the iterations by repeatedly processing a set of points fed from a FIFO buffer. This would reduce the size of the system considerably, allowing it to fit in smaller FPGA platforms and to perform more iterations, at the expense of longer processing times.

- Currently the size of the image region W computed in the Normalize Window module prevents edge effects on a 3×3 neighbourhood centred at $\mathbf{x}_w$ (Section 3.4.1). For this reason, only changes in location of up to 1 pixel can be reliably computed in each iteration. Increasing the size of W would allow larger changes in location per iteration, which could lead to faster convergence rates. However, this would also require more pixels to be retrieved from the image

buffer to implement the bilinear interpolation, which increases the number of clock cycles spent processing each feature.

- The system can be modified to process images of sizes other than 640×480 pixels. This should be a reasonable task considering that the One Iteration module, which is the largest and most complex part of the circuit, does not depend on the size of the image. Small modifications are required in the Multiscale Harris stage, which include changing the length of the shift registers used to align the image and the gaussian coefficients for filtering (currently set to 640 pixels). Moreover, because the images that arrive from the camera and the images that are sent to the monitor are stored in frame buffers before and after the detector circuit, images of other sizes can be read and written to these buffers with only minor modifications to the design, as long as the video standards (NTSC and VGA) are not changed.

- The feature detector circuit is independent of the video interfaces that provide the images to be processed. Therefore, it is possible to modify the Video Input module to accept video in standards other that NTSC without modifying the processing stages (i.e., Multiscale Harris and One Iteration modules). The Transmogrifier-4 already includes video interfaces for the IEEE 1394 standard, commonly known as *Firewire* video, which can achieve data transfer rates of up to 400Mbps (in 1394a) and 800Mbps (in 1394b).

This implementation of the Harris-Affine feature detector shows that it is possible to create a smart camera system that includes a complex feature detection stage given the amount of resources available in current FPGA devices. It is the opinion of the author that given the rapid increase in the density and speed of FPGAs and the importance of robust feature detection in computer vision systems, FPGA-based feature detectors will become increasingly popular as integral parts of vision systems developed in academia and industry.

Appendix A

Cramer's rule

Let A be an $n \times n$ matrix and suppose that $\det A \neq 0$. Then the unique solution to the system $A\mathbf{x} = \mathbf{b}$ is given by [20]

$$x_1 = \frac{D_1}{D},\ x_2 = \frac{D_2}{D},\ \ldots,\ x_n = \frac{D_n}{D} \tag{A.1}$$

where $D = \det A$, $D_i = \det A_i$ and A_i is the matrix obtained by replacing the ith column of A with $\mathbf{b}$.

Appendix B

Computation of the eigenvalues of μ and U

Given a 2×2 matrix A, its eigenvalues λ_1 and λ_2 are the roots of the characteristic equation

$$\begin{aligned} \det(A - \lambda I) &= \det\left(\begin{bmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\ &= \lambda^2 - \lambda(A_{11} + A_{22}) + A_{11}A_{22} - A_{12}A_{21} \\ &= 0 \end{aligned} \tag{B.1}$$

Solving for the roots gives

$$\begin{aligned} \lambda_1 &= \frac{1}{2}\left(A_{11} + A_{22} + \sqrt{(A_{11} + A_{22})^2 - 4(A_{11}A_{22} - A_{12}A_{21})}\right) \\ &= \frac{1}{2}\left(A_{11} + A_{22} + \sqrt{(A_{11} - A_{22})^2 + 4A_{12}A_{21}}\right) \end{aligned} \tag{B.2}$$

and

$$\lambda_2 = \frac{1}{2}\left(A_{11} + A_{22} - \sqrt{(A_{11} - A_{22})^2 + 4A_{12}A_{21}}\right) \tag{B.3}$$

The eigenvalues of μ and U can be computed in a similar manner.

In the specific case of the eigenvalues of μ used to compute the isotropy ratio (Section 3.4.3), we used the fact that $\mu_{12} = \mu_{21}$ to replace the square root in equations B.2

and B.3 by a sum of absolute values

$$\begin{aligned}\lambda(A) &= \frac{1}{2}\left(A_{11}+A_{22}\pm\sqrt{(A_{11}-A_{22})^2+4\,{A_{21}}^2}\right)\\ &\approx \frac{1}{2}\left(A_{11}+A_{22}\pm\left(|A_{11}-A_{22}|+4\,|A_{21}|\right)\right)\end{aligned}$$

therefore

$$\lambda(\mu)\approx\frac{1}{2}\left(L_x^{2\prime}+L_y^{2\prime}\pm\left(|L_x^{2\prime}-L_y^{2\prime}|+4\,|L_xL_y{}'|\right)\right) \tag{B.4}$$

This approximation provides values that are sufficiently close to the actual eigenvalues for the purpose of computing the isotropy ratio and that do not require the use of a square root function, which is costly in hardware. In addition, the factor σ_D^2 in the expression of μ (Equation 3.3) was not included in the calculation of the eigenvalues, since it is cancelled during the evaluation of Q.

Appendix C

Computation of the inverse square root of μ_{new}

The algorithm presented in [37] updates the shape matrix U by multiplying the current value of U with the inverse square root of the second moment matrix μ_{new}. In general, computing the inverse square root of a matrix is an expensive operation that may require an iterative procedure. In the case of the second moment matrix, however, we can simplify the operation using the fact that μ_{new} is a real symmetric matrix. For this purpose, we take into account the following theorems of linear algebra:

1. An $n \times n$ matrix A is *orthogonally diagonalizable* if there exists an orthogonal matrix Q such that $Q^t A Q = D$ where $D = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$ and $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of A. Furthermore, A is orthogonally diagonalizable if and only if A is symmetric. [20]

2. If $A \in \mathbb{C}^{n \times n}$ and $A = X\ diag(\lambda_1(A) \ldots \lambda_n(A))\ X^{-1}$, then $f(A) = X\ diag(f(\lambda_1), f(\lambda_2)...., f(\lambda_n))\ X^{-1}$. [19]

μ_{new} is a 2×2 symmetric matrix and therefore it is diagonalizable. The orthogonal matrix Q is formed by eigenvectors $\mathbf{v}_1 = [v_{11}, v_{12}]^T$ and $\mathbf{v}_2 = [v_{21}, v_{22}]^T$ of μ_{new}, such

that $Q = [\mathbf{v}_1, \mathbf{v}_2]$. The eigenvectors can be found by calculating the eigenvalues of $\lambda_1(\mu_{new})$ and $\lambda_2(\mu_{new})$ of μ_{new} from equations B.2 and B.3, and solving the system of equations

$$\mu_{new} \cdot \mathbf{v}_1 = \lambda_1 \cdot \mathbf{v}_1, \tag{C.1}$$

$$\mu_{new} \cdot \mathbf{v}_2 = \lambda_2 \cdot \mathbf{v}_2, \tag{C.2}$$

and

$$||\mathbf{v}_1|| = ||\mathbf{v}_2|| = 1 \tag{C.3}$$

In the particular case where $\lambda_1(\mu_{new}) = \lambda_2(\mu_{new})$, the eigenvectors can be chosen to be any pair of orthogonal vectors that preserve the sign of the eigenvalues in equations C.1 and C.2, such as $\mathbf{v}_1 = [1, 0]^T$ and $\mathbf{v}_2 = [0, 1]^T$.

Solving for the first eigenvector in terms of the entries of μ_{new} gives

$$v_{11} = \frac{\sqrt{{\mu_{21}}^2}}{\sqrt{\mu_{21} + (\lambda_1 - \mu_{11})^2}} \qquad (v_{11} \geq 0) \tag{C.4}$$

and

$$v_{12} = v_{11} \cdot \frac{(\lambda_1 - \mu_{11})}{\mu_{21}} \tag{C.5}$$

where

$$\mu_{new} = \begin{bmatrix} \mu_{11} & \mu_{21} \\ \mu_{12} & \mu_{22} \end{bmatrix} \tag{C.6}$$

We form $\mathbf{v}_2 = [v_{12}, -v_{11}]$ orthogonal to $\mathbf{v}_1$, such that the sign of λ_2 is preserved in Equation C.2.

From theorem 2, the inverse square root matrix of μ_{new} can be calculated as

$$\begin{aligned} {\mu_{new}}^{-\frac{1}{2}} &= Q\ diag(\lambda_1(\mu_{new})^{-\frac{1}{2}}, \lambda_2(\mu_{new})^{-\frac{1}{2}})\ Q^T \\ &= \begin{bmatrix} v_{11} & v_{21} \\ v_{12} & v_{22} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{\lambda_1}} & 0 \\ 0 & \frac{1}{\sqrt{\lambda_2}} \end{bmatrix} \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \end{bmatrix} \\ &= \begin{bmatrix} \frac{{v_{11}}^2}{\sqrt{\lambda_1}} + \frac{{v_{12}}^2}{\sqrt{\lambda_2}} & \frac{v_{11}v_{12}}{\sqrt{\lambda_1}} - \frac{v_{11}v_{12}}{\sqrt{\lambda_2}} \\ \frac{v_{11}v_{12}}{\sqrt{\lambda_1}} - \frac{v_{11}v_{12}}{\sqrt{\lambda_2}} & \frac{{v_{12}}^2}{\sqrt{\lambda_1}} + \frac{{v_{11}}^2}{\sqrt{\lambda_2}} \end{bmatrix} \end{aligned} \tag{C.7}$$

where we have made use of the fact that $Q^{-1} = Q^T$. Replacing expressions C.4 and C.5 in C.7 gives ${\mu_{new}}^{-\frac{1}{2}}$ in terms of the elements and eigenvalues of μ_{new}:

$$ {\mu_{new}}^{-\frac{1}{2}} = \begin{bmatrix} \frac{{\mu_{21}}^2}{\sqrt{\lambda_1}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} + \frac{(\lambda_1-\mu_{11})^2}{\sqrt{\lambda_2}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} & \frac{(\lambda_1-\mu_{11})\mu_{21}}{\sqrt{\lambda_1}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} - \frac{(\lambda_1-\mu_{11})\mu_{21}}{\sqrt{\lambda_2}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} \\ \frac{(\lambda_1-\mu_{11})\mu_{21}}{\sqrt{\lambda_1}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} - \frac{(\lambda_1-\mu_{11})\mu_{21}}{\sqrt{\lambda_2}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} & \frac{(\lambda_1-\mu_{11})^2}{\sqrt{\lambda_2}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} + \frac{{\mu_{21}}^2}{\sqrt{\lambda_1}\,({\mu_{21}}^2+(\lambda_1-\mu_{11})^2)} \end{bmatrix} \tag{C.8} $$

In this form, the elements of ${\mu_{new}}^{-\frac{1}{2}}$ have common factors, which allows for a more efficient implementation since the factors can be computed once and then combined to form the different elements.

Appendix D

Canny edge detector

The Canny edge detector [9] computes the gradient of a smoothed version of the input image and determines whether a pixel belongs to an edge based on the magnitude and direction of the gradient at the pixel location. Figure D.1 shows the main stages of the detector.

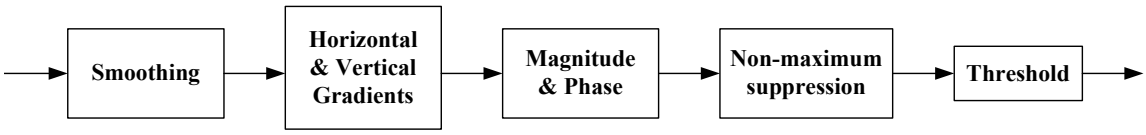


Figure D.1: Block diagram of the Canny edge detector

Implementations of the edge detector in software cannot exploit the relative independence of the various processing stages. In a hardware implementation on the other hand, stages can be pipelined to process frames from a video sequence in real time.

In this project, a pipelined architecture of the system was designed based on the architecture described in [55]. The system was implemented on a Xilinx Virtex-II Multimedia board, where it is capable of processing 640×480 frames at 30 frames per second using a fraction of the capacity of the FPGA. The system was later ported to an Altera DE2 board and the Transmogrifier-4.

D.1 Hardware architecture

The smoothing stage consists of convolving the input image with the 5×5 mask

$$g(x,y) = \frac{1}{128} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{D.1}$$

This mask approximates a Gaussian kernel of standard deviation equal to 1.4 pixels. Similarly, the gradients I_x and I_y are computed by filtering the output of the smoothing stage with the kernels

$$\frac{d}{dx} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \quad \text{and} \quad \frac{d}{dy} = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{D.2}$$

The convolution operations in the smoothing and gradient stages are implemented using an architecture similar to the one described in Section 3.3.2, which includes a set of delays used to align the pixels in the current image window with the kernel coefficients before the arithmetic operations are performed.

The magnitude of the image gradient is calculated from I_x and I_y by using a simplified expression that avoids the use of the expensive square root operator

$$\text{magnitude} = \sqrt{{I_x}^2 + {I_y}^2} \approx |I_x| + |I_y| \tag{D.3}$$

The gradient direction is chosen from the signs and relative magnitudes of I_x and I_y. Figure D.2 shows how the gradient direction is quantized into eight possible directions.

The non-maximum suppression stage compares the gradient magnitude at each pixel location with the average gradient magnitudes along the direction of the gradient. In the example shown in Figure D.3, the average gradient magnitudes along

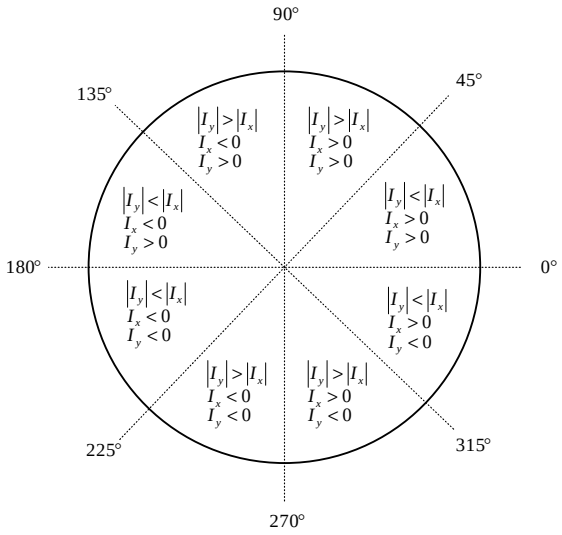


Figure D.2: Gradient orientations

the gradient direction are the magnitudes at points a and b. These are computed as the simple average of the gradients at points $(x-1, y-1)$ and $(x, y-1)$ (for a), and $(x, y+1)$ and $(x+1, y+1)$ (for b). A pixel is discarded unless its gradient magnitude is greater or equal than both average gradients. In this way, the system suppresses responses that are locally weak.

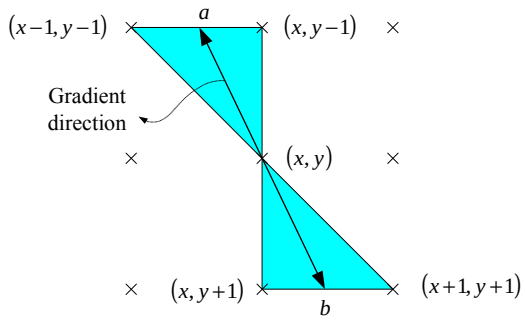


Figure D.3: Average gradients along gradient direction

The last stage implements a threshold operation that includes hysteresis. A pixel

whose gradient (after the non-maximum suppression stage) is above a higher threshold T_H is accepted as an edge pixel, a pixel with gradient below a lower threshold T_L is discarded, and a pixel whose gradient is in the range $[T_L, T_H]$ is discarded unless at least one of the pixels in its 8-point neighbourhood has a gradient magnitude above T_H. Hysteresis allows edge pixels to be connected into continuous edges in the presence of small local disturbances.

Figure D.4 shows the output of the fixed-point MATLAB implementation used to model the hardware circuit.

Figure D.4: Output of the Canny edge detector

Bibliography

[1] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. Pyramid method in image processing. *RCA Engineer*, 29(6):33–41, 1984.

[2] L. Alvarez and F. Morales. Affine morphological multiscale analysis of corners andmultiple junctions. *Internaltion Journal of Computer Vision*, 25(2):95–107, 1997.

[3] A. Baumberg. Reliable feature matching across widely separated views. In *Conference on Computer Vision and Pattern Recognition*, pages 1774–1781, Hilton Head, SC, USA, June 2000.

[4] A. Benedetti and P. Perona. Real-time 2-d feature detection on a reconfigurable computer. In *Conference on Computer Vision and Pattern Recognition*, pages 586–593, 1998.

[5] A. Bissacco and S. Ghiasi. Fast visual feature selection and tracking in a hybrid reconfigurable architecture. In *2nd Workshop on Applications of Computer Vision, ECCV 2006*, Gratz, May 2006.

[6] C.-S. Bouganis, P. Y. K. Cheung, J. Ng, and A. A. Bharath. A steerable complex wavelet construction and its implementation on FPGA. In *14th International Conference on Field Programmable Logic and Applications*, volume 3203 of *Lecture Notes in Computer Science*, pages 394–403, January 2004.

[7] L. Bretzner and T. Lindeberg. Feature tracking with automatic selection of spatial scales. *Computer Vision and Image Understanding*, 71(3):385–392, September 1998.

[8] C. Cabani and W. J. MacLean. A proposed pipelined-architecture for FPGA-based affine-invariant feature detectors. *The Second IEEE Workshop on Embedded Computer Vision, (CVPR 2006)*, June 2006.

[9] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[10] J. L. Crowley and A. C. Parker. A representation for shape based on peaks and ridges in the difference of low-pass transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(2):156–170, March 1984.

[11] A. Darabiha, J. Rose, and W. J. MacLean. Reconfigurable hardware implementation of a phase-correlation stereo algorithm. *Machine Vision and Applications*, 17(2):116–132, 2006.

[12] F. Dellaert and S. Tariq. A multi-camera pose tracker for assisting the visually impaired. In *1st IEEE Workshop on Computer Vision Applications for the Visually Impaired*, 2005.

[13] Development and Education Board (DE2). [Online]. Available http://www.altera.com/education/univ/materials/boards/unv-de2-board.html.

[14] J. Fender and D. Galloway. Simple_ddr_video. Unpublished reference design, 2006.

[15] P. Fiore, D. Kottke, and D. Gampagna. Efficient feature tracking with application to camera motion estimation. In *Conference Record of the 32nd Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 949–953, November 1998.

[16] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2002.

[17] D. Galloway. [Online]. Available http://www.eecg.toronto.edu/~tm4/ports.pdf.

[18] P. Giacon, S. Saggin, G. Tomasi, and M. Busti. Implementing DSP algorithms using Spartan-3 FPGAs. *Xcell Journal*, Issue 53:22–25, 2005.

[19] G. H. Golub and C. F. Van Loan. *Marix Computations*. The Johns Hopkins University Press, second edition, 1989. Pages 540–542.

[20] S. I. Grossman. *Elementary Linear Algebra*. Saunders College Publishing, fourth edition, 1991. Pages 147–148, 414–415.

[21] C. Harris and M. Stephens. A combined corner and edge detector. In *Proceedings Fourth Alvey Vision Conference*, pages 147–151, Manchester, United Kingdom, 1988.

[22] R. Hartley and A. Zisserman. Multiple view geometry in computer vision - figures. [Online]. Available http://www.robots.ox.ac.uk/~vgg/hzbook /HZfigures.html. Retrieved July 30, 2006.

[23] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[24] A. Honarvar and C. Cabani. Canny edge detector. Project Report. VLSI Systems Design ECE1373. Department of Computer and Electrical Engineering. University of Toronto, July 2005.

[25] M. Leeser, S. Miller, and H. Yu. Smart camera based on reconfigurable hardware enables diverse real-time applications. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages 147–155, 2004.

[26] T. Lindeberg. Detecting salient blob-like image structures and their scales with a scale-space primal sketch: A method for focus-of-attention. *International Journal of Computer Vision*, 11(3):283–318, December 1993.

[27] T. Lindeberg. Scale-space theory: A basic tool for analysing structures at different scales. *Journal of Applied Statistics*, 21(2):224–270, 1994.

[28] T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision*, 30(2):77–116, 1998.

[29] T. Lindeberg and J. G. årding. Shape-adapted smoothing in estimation of 3-d shape cues from affine deformations of local 2-d brightness structure. *Image and Vision Computing*, 15(6):415–434, June 1997.

[30] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision ICCV*, pages 1150–1157, Corfu, Greece, 1999.

[31] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *nternational Journal of Computer Vision*, 60(2):91–110, 2004.

[32] W. J. MacLean. An evaluation of the suitability of FPGAs for embedded vision systems. In *The First IEEE Workshop on Embedded Computer Vision, CVPR 2005*, New York, June 2005.

[33] D. K. Masrani and W. J. MacLean. A real-time large disparity range stereo-system using FPGAs. In *7th Asian Conference on Computer Vision*, volume 3852 of *Lecture Notes in Computer Science*, pages 42–51, 2006.

[34] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761–767, September 2004.

[35] K. Mikolajczyk and C. Schmid. Indexing based on scale invariant interest points. In *IEEE International Conference on Computer Vision*, pages 525–531, 2001.

[36] K. Mikolajczyk and C. Schmid. An affine invariant interest point detector. In *European Conference on Computer Vision*, volume 4, pages 128–142, 2002.

[37] K. Mikolajczyk and C. Schmid. Scale & affine invariant interest point detectors. *International Journal of Computer Vision*, 60(1):63–86, 2004.

[38] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 27(10):1615–1630, 2005.

[39] ModelSim. [Online]. Available http://www.model.com.

[40] H. Moravec. Rover visual obstacle avoidance. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 785–790, August 1981.

[41] D. Nguyen, D. Halupka, P. Aarabi, and A. Sheikholeslami. Real-time face detection and lip feature extraction using field-programmable gate arrays. *IEEE Transactions on Systems, Man and Cybernetics*, 36(4):902–912, August 2006.

[42] Quartus II Software. [Online]. Available http://www.altera.com/products /software/products/quartus2/qts-index.html.

[43] F. Schaffalitzky and A. Zisserman. Multi-view matching for unordered image sets, or "How do I organize my holiday snaps?". In *Proceedings of the 7th European Conference on Computer Vision, Copenhagen, Denmark*, volume 1, pages 414–431, 2002.

[44] J. Schlessman, C.-Y. Chen, W. Wolf, B. Ozer, K. Fujino, and K. Itoh. Hardware/software co-design of an FPGA-based embedded tracking system. *2nd Workshop on Embedded Computer Vision, CVPR 2006)*, 2006.

[45] C. Schmid, R. Mohr, and C. Bauckhage. Evaluation of interest point detectors. *International Journal of Computer Vision*, 37(2):151–172, 2000.

[46] S. Se, T. Barfoot, and P. Jasiobedzki. Visual motion estimation and terrain modeling for planetary rovers. In *8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Munich, 2005.

[47] M. Sen, I. Corretjer, F. Haim, S. Saha, S. S. Bhattacharyya, J. Schlessman, and W. Wolf. Computer vision on FPGAs: Design methodology and its application to gesture recognition. *1st Workshop on Embedded Computer Vision, CVPR 2005*, 2005.

[48] A. Shokoufandeh, I. Marsic, and S. Dickinson. View-based object recognition using saliency maps. *Image and Vision Computing*, 17:445–460, 1999.

[49] Stratix Device Handbook. [Online]. Available http://www.altera.com/literature /hb/stx/stratix_handbook.pdf.

[50] The MathWorks: Matlab. [Online]. Available http://www.mathworks.com /products/matlab.

[51] The Transmogrifier-4 Project. [Online]. Available http://www.eecg.toronto.edu /∼tm4.

[52] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.

[53] T. Tuytelaars and L. Van Gool. Content-based image retrieval based on local affinely invariant regions. *Proceedings of the Third International Conference on Visual Information Systems*, pages 493–500, 1999.

[54] T. Tuytelaars and L. Van Gool. Wide baseline stereo based on local, affinely invariant regions. *Proceedings of the 11th British Machine Vision Conference*, pages 412–422, 2000.

[55] D. Venkateshwar Rao and M. Venkatesan. An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, volume 2, pages 843–847, 2004.

[56] Vienna University of Technology. Computer Science Department. [Online]. Available http://www.prip.tuwien.ac.at/Research/ImagePyramids.

[57] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[58] Xilinx ISE. [Online]. Available http://www.xilinx.com/support/sw_manuals/xilinx6/index.htm.