MONASH University

FPGA Acceleration of Multilevel ORB Feature Extraction for Computer Vision

Joshua Weberruss

BCSE(Hons), DipLang(German)

A thesis submitted for the degree of Doctor of Philosophy at Monash University in 2018

Department of Electrical and Computer Systems Engineering

0.1 Copyright notice

©Joshua Weberruss 2018

I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.

0.2 Abstract

Feature matching is a fundamental step in many image processing tasks, such as panorama stitching and stereo depth estimation. In essence it involves finding some characteristic in multiple images and then identifying commonalities between these detected characteristics. For instance, one could locate blobs of colour in two different images, then identify pairs of blobs of the same colour in the two images.

A number of algorithms have been developed, both for detecting features, and also for matching those features once detected. This work looks into the ORB[1] feature extraction algorithm, and specifically investigates its applicability to hardware acceleration using a Field-Programmable Gate Array (FPGA) device.

This work has designed and developed an FPGA implementation of the Harris–Stephens corner detection algorithm and ORB feature extraction algorithm. The system operates using an image pyramid, allowing detection and matching of corners at different scales, and has been designed with the foremost goal being high throughput.

The system has been developed in SystemVerilog, and each module is accompanied by a test-bench to verify the correct operation of the hardware designs.

The developed system has been implemented and tested on an Arria V FPGA, and attains very high framerates on real data: approximately 72 fps at 1920×1080 and 488 fps at 640×480, compared to a quad-core CPU with 13 fps and 88 fps respectively. The power consumption under load is also very low, drawing approximately 6 W, in comparison to a CPU at around 90W. The full hardware accelerator, which has been named a *Tarsier*, can be attached to a PC via PCI-Express bus, and communicates with userspace software by way of a Linux kernel-mode driver.

In order to achieve tight realtime bounds on computation time, and to best take advantage of the FPGA platform, the algorithms have been modified slightly from their CPU originals. This thesis contains analysis of the effects of these modifications, and demonstrates that the changes have a minimal effect on the utility of the feature extraction.

[1] Oriented FAST (Features by Accelerated Segment Test) and Rotated BRIEF (Binary Robust Independent Elementary Features)

0.3 Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution, and, to the best of my knowledge and belief, this thesis contains no material previously published or written by any other person, except where due reference is made in the text of the thesis.

Signature	
Print name	Joshua Weberruss
Date	29/06/2018

0.4 Publications

- J. Weberruss, L. Kleeman and T. Drummond, 'ORB Feature Extraction and Matching in Hardware', in *Proceedings of the Australiasian Conference on Robotics and Automation*, Australian Robotics and Automation Association, 2015
- J. Weberruss, L. Kleeman, D. Boland and T. Drummond, 'FPGA acceleration of multilevel ORB feature extraction for computer vision', in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056856

0.5 Acknowledgements

- This research was supported by an Australian Government Research Training Program (RTP) Scholarship.

- This research was supported by the Australian Research Council Centre of Excellence for Robotic Vision (project number CE140100016)

- The author would particularly like to thank his supervisors, Lindsay Kleeman (main) and Tom Drummond (associate), and his co-author, David Boland, for their invaluable assistance over the duration of this PhD

Contents

0.1 Copyright notice 1

0.2 Abstract 2

0.3 Declaration 3

0.4 Publications 4

0.5 Acknowledgements 5

Contents **6**

1 Introduction **11**

1.1 Objectives of this Research 14

1.2 Contributions of this Work 15

1.3 Thesis Outline 16

2 Feature Detection **17**

2.1 Introduction 17

2.1.1 Images 17

2.1.2 Features 18

2.2 Algorithms 20

2.2.1 Harris–Stephens 20

2.2.2 FAST Corners 24

2.2.3 Non-Maximum Suppression 25

2.3 Hardware Primitives 27

2.3.1 Sliding Window 27

2.3.2 Configurable Linebuffer 29

2.3.3 Margins 30

2.4 Previous Work . . . 36
2.4.1 Harris–Stephens . . . 36
2.5 Implementation . . . 39
2.5.1 Harris–Stephens Detector . . . 40
2.5.2 Non-Maximum Suppression . . . 45
2.5.3 Performance . . . 45
2.6 Conclusion . . . 48

3 Feature Descriptor Extraction 50

3.1 Introduction . . . 50
3.1.1 Feature Descriptors and Extractors . . . 50
3.1.2 Invariance . . . 51
3.2 Algorithms . . . 53
3.2.1 SIFT . . . 53
3.2.2 BRIEF . . . 54
3.2.3 ORB . . . 54
3.3 Previous Work . . . 60
3.3.1 SIFT . . . 60
3.3.2 BRIEF . . . 60
3.3.3 ORB . . . 61
3.4 Implementation . . . 63
3.4.1 Feature Orientation . . . 64
3.4.2 ORB Window . . . 75
3.4.3 ORB Module . . . 78
3.4.4 ORB Arbitrator . . . 80
3.4.5 Design Choices . . . 80
3.5 Other Work . . . 84
3.6 Feature Descriptor Matching . . . 86
3.6.1 Thresholding and Prominence Matching . . . 86
3.7 Conclusion . . . 88
3.8 Comparison with other work . . . 89

3.8.1 Scale invariance . . . 89

4 Multiscale Processing **91**

4.1 Introduction . . . 91

4.1.1 Image Pyramid . . . 91

4.1.2 Resampling . . . 93

4.1.3 Interpolation Filters . . . 95

4.2 Implementation . . . 98

4.2.1 Pyramid Structure . . . 98

4.2.2 Resampling . . . 101

4.2.3 FIR Filter . . . 103

4.2.4 Dimensions . . . 104

4.2.5 Coordinate Transformation . . . 105

4.3 Conclusion . . . 106

5 Analysis, Validation and Integration **107**

5.1 Introduction . . . 107

5.1.1 Simulation tools . . . 108

5.2 Pyramid Structure . . . 110

5.3 Harris–Stephens Parameters . . . 117

5.4 Feature Drop Analysis . . . 121

5.4.1 Introduction . . . 121

5.4.2 Guide to Timing Diagrams . . . 123

5.4.3 Strategy 0: No buffering . . . 124

5.4.4 Strategy 1: Stalling when all modules are processing . . . 125

5.4.5 Strategy 2: Stalling when any module is processing . . . 125

5.4.6 Small-Scale Analysis . . . 126

5.4.7 Large-Scale Analysis . . . 130

5.4.8 Conclusion . . . 141

5.5 Testbenching . . . 142

5.6 *Tarsier*: Integrating Corner Detection, ORB Feature Extraction, Scaling and Buffering . . . 143

5.6.1 Input . . . 143
5.6.2 Output . . . 144
5.6.3 Integration . . . 144
5.6.4 Performance . . . 145
5.6.5 Testing . . . 147
5.7 Conclusion . . . 149

6 Generalisation, Conclusions and Future Work 150

6.1 Generalisation of Results: Stencil Codes . . . 151
6.1.1 Discrete-Domain Problems . . . 152
6.1.2 Continuous-Domain Problems . . . 153
6.1.3 Boundary Conditions . . . 153
6.1.4 FPGA Realisation . . . 154
6.1.5 Conclusion . . . 155
6.2 Conclusions . . . 156
6.3 Future Work . . . 157
6.3.1 Hardware matching . . . 157
6.3.2 Further detection and scaling analysis . . . 157
6.3.3 Further drop analysis . . . 157
6.3.4 Comparison with optimised GPU implementation . . . 158
6.3.5 Separate clock domains . . . 158
6.3.6 Tiling . . . 158

Appendices 160

A Overview of Hardware Description Language Modules 161

A.1 Concepts . . . 162
A.2 AdderTreePipelined . . . 163
A.3 BufferedCornersAndDescriptors . . . 164
A.4 CoordinateTransformer . . . 166
A.5 CornersAndDescriptors . . . 167
A.6 DimensionCalculator_4_5 . . . 170

A.7 FIFO . . . 172
A.8 HarrisCornersAndNonmax . . . 173
A.9 HarrisCornersPipelined . . . 175
A.10 HarrisMatrixPipelined . . . 177
A.11 MultilevelBarrier . . . 178
A.12 MultitapShiftRegister . . . 179
A.13 NonmaxSuppression . . . 181
A.14 ORB2 . . . 182
A.15 ORBArbitrator . . . 184
A.16 ORBMultiscale . . . 186
A.17 ORBWindow . . . 189
A.18 RAMBlock . . . 191
A.19 RecBinaryMux . . . 192
A.20 Scale_1_2_Bilinear . . . 193
A.21 Scale_4_5_Bilinear . . . 195
A.22 SectorSel . . . 197
A.23 ShiftRegister . . . 198
A.24 SlidingWindow . . . 199
A.25 SlowDividerUnsigned . . . 200
A.26 UnsignedAdderTreePipelined . . . 201
A.27 VectorRotate . . . 202

Chapter 1

Introduction

Many robots today use GPS as their primary source of nagivation information, such as unmanned aerial vehicles, autonomous vehicles, and agricultural robots. However, when indoors or when GPS is unavailable, or too inaccurate for a robot to use, many robots turn to visual sensors for localisation and mapping.

We humans use visual information as our primary source when navigating. Our stereoscopic, binocular vision allows us to estimate the distance to objects based on the disparity between the two images we receive, one from each eye, and vision allows us to determine our position in space by tracking how objects around us move. The same principles are applicable to robots with cameras mounted atop them.

Cameras take in light from the environment around a robot and convert the information in this light into images, which can be encoded electrically and used for robot navigation. The images are processed to identify objects, and these objects can be tracked by a computer onboard the robot to estimate the robot's motion accurately. This processing, however, can be computationally demanding, and consumes significant amounts of energy. For a mobile vehicle, energy is often at a premium, and every joule saved from computation can be spent doing other tasks, such as driving locomotion.

This work seeks to design and develop a hardware accelerator for this image processing. A hardware accelerator, implemented using a Field-Programmable Gate Array (FPGA), is able to perform the image processing extremely quickly, significantly faster than a CPU, while using much less power. In addition, while the hardware accelerator is processing an image, the CPU is free to do other work, potentially allowing the use of a cheaper or less powerful CPU.

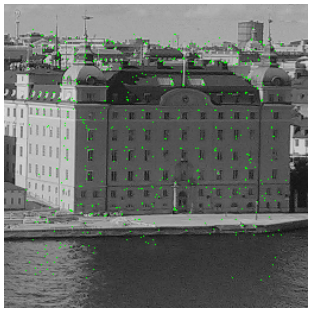


Figure 1.1: Features (green) detected in an image

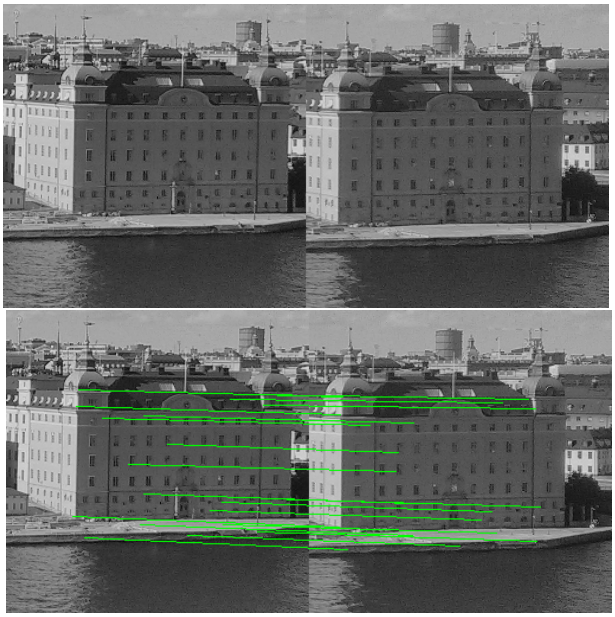


Figure 1.2: Matches (green) detected in an image pair

The approach used in this work is called ***sparse feature detection, extraction and matching***. This means that interesting points in the images (***features***) are located (***detection***), and then a value is derived from the features (***extraction***) which allows them to be ***matched*** against other features in other images. The features only use interesting regions of the images, discarding areas with low information, such as flat colours, so the coverage of the images is ***sparse***.

Figure 1.1 shows an image with features detected, shown in green. These features are located at points that are determined to be corners by the Harris–Stephens corner detector [3].

It is useful to be able to recognise the same physical object in different images. Feature matching is able to assist with this, by finding features in two images that are similar, and recording the ***correspondences***. Figure 1.2 shows two images of the same scene, taken from different viewpoints. The features are detected using Harris–Stephens, and then ***feature descriptors*** are

extracted using the ORB feature descriptor extraction algorithms [4]. These descriptors allow comparisons to be made between features, and similarity measured. Features whose descriptors are more similar than a threshold are considered matches, and these matches are shown in figure 1.2. Green lines connect points in each image that look similar, according to the difference between their descriptors.

Over the course of this PhD candidature, a hardware acceleration device, termed a *Tarsier*, has been developed. This device is implemented using an Arria V GX Starter Kit FPGA board, and connects to a host PC via PCI-express. Images may be streamed to the device over the PCI-express interconnect, where they are processed, and the results are returned to the host PC over the same interconnect.

The *Tarsier* accelerator has been integrated into the ORB-SLAM2 [5], where it is used to accelerate feature detection and matching. ORB-SLAM2 is a software implementation of ***Visual Simultaneous Localisation And Mapping (Visual SLAM)***. SLAM is a family of algorithms for taking observational input from the environment of a robotic vehicle and using the information contained therein to construct a map of the robot's surroundings, and localise the robot within that map.

Visual SLAM algorithms take images from cameras and use these to build a map of the surroundings visible to the camera. SLAM is, in general, a hard computational problem, and robots making use of visual SLAM are far inferior to humans performing the same tasks.

The device implements the feature detection and extraction phases of the image processing, but the matching is performed on the host PC using the CPU. An earlier version of the hardware architecture, developed and presented in [1], did include a basic matching system, but this was removed in the process of developing the integration with ORB-SLAM2, since it implements a more sophisticated matching process.

1.1 Objectives of this Research

1. To develop a hardware accelerator that can operate with hard ***real-time*** constraints to detect features in images and extract descriptors for them. In this instance, real-time means that execution time is constant for a given image size. The hardware accelerator should be easily usable from within program code running on a host machine.

2. To maximise the frame rate the accelerator is able to work at. The frame rate should be at least 60 frames per second (fps) at 1920×1080 resolution.

3. As far as possible, to minimise the modification to the standard algorithms, while still achieving real-time guarantees and high frame rate.

4. To characterise the performance of the accelerator, and determine how variation in the design parameters affects the quality and usefulness of the output. This will increase the applicability of the design to different hardware platforms.

5. To release the source code openly, so that the research community can build on this work.

1.2 Contributions of this Work

The primary contributions of this work are:

1. Development of a new hardware architecture for accelerating the detection and description of ORB feature in real time using FPGAs. This work [1], while not the first, makes more effective use of the FPGA platform than the preceding work [6], and thus demonstrates significantly improved speed. It predates two other publications [7], [8], while remaining competitive.

2. Development and analysis of a multi-scale ORB implementation in hardware [2] using a streaming image pyramid structure. This work was the first and remains the only such multi-scale ORB implementation in hardware, and it results in significantly improved matching performance. Other ORB implementations [6]–[8] only perform feature detection and extraction at the base image scale.

3. Characterisation of the performance of the system with varying image pyramid structures and numbers of modules per level. This allows customisation of the design based on the hardware budget available and desired performance.

4. Characterisation of the number of features that must be dropped as a result of the hard real-time constraints of the system, and thus not appear in the output. This is highly dependent on the use of buffering in the ORB feature extraction pipeline.

5. Development of a Linux kernel-mode driver to interface with the *Tarsier* device over PCI-express

6. Provision of an open-source, Mozilla Public Licensed repository of the SystemVerilog code, to allow effective comparison by the research community (`https://github.com/raiker/tarsier`).

1.3 Thesis Outline

This thesis is divided into a number of chapters, with three chapters addressing design and implementation of the FPGA system, one chapter addressing analysis of the design and exploration of the design parameter space, and one presenting conclusions and future work.

Chapter 2 examines feature detection. The state of the art is investigated, previous hardware implementations are described and compared, and the implementation of the Harris–Stephens feature detector used in this work is presented.

Chapter 3 examines feature extraction. The chapter investigates a number of different approaches to feature extraction, then describes the existing work in hardware acceleration of the selected algorithm, ORB. The implementation of ORB used in this work is presented in detail, and it is compared to other implementations in the literature.

Chapter 4 examines multiscale processing, the approach of applying feature detection and extraction on an image at multiple scales simultaneously. The chapter investigates why multiscale processing is useful, then presents the efficient streaming architecture employed in this work.

Chapter 5 examines the parameter space of the algorithms implemented on the FPGA, and investigates the effects that changes in one parameter can have on the system performance. The validation system used to verify the accuracy of the hardware is described, and then the integration of the *Tarsier* accelerator into ORB-SLAM2 is examined in depth. The performance of the *Tarsier* device is discussed in this chapter.

Chapter 6 contains the conclusions drawn from the project, as well as an investigation of how the work presented can be generalised to other problems, before closing with work for the future.

Appendix A contains documentation for each of the hardware modules making up the core of the *Tarsier* device. The SystemVerilog code for the *Tarsier* core is available under a Mozilla Public Licence on GitHub at `https://github.com/raiker/tarsier`.

Chapter 2

Feature Detection

2.1 Introduction

Feature detection is the process of finding interesting features in images. This is often the first step in a computer vision system, since it identifies information-rich areas of the image for further processing, enabling effective use of limited computation resources. In this system, once detected, feature locations are passed to a feature extractor (see chapter 3).

This chapter opens with a definition of images and of features, then investigates several feature detection algorithms in current use. A description of the implementation of feature detection in this system follows.

2.1.1 Images

An ***image*** is a two-dimensional rectangular grid of picture elements, or ***pixels***. The pixels in an image can be colour (three-dimensional) or greyscale (one-dimensional). This work deals solely with greyscale images. Mathematically, the pixels are discrete samples from a continuous 2-dimensional image signal, termed $I(x, y)$. This formal representation becomes significant when discussing image downsampling, in section 4.1.2.

The value of a greyscale pixel is alternatively referred to as the ***intensity*** at that point. The pixel intensity represents the value of the underlying image signal at that coordinate.

$$Image[x, y] = I(x, y), \text{ where } x, y \in \mathbb{N}^0 \tag{2.1}$$

The spatial derivatives of image intensity are commonly used in computer graphics to determine local properties of the image. Since we have only discrete samples into a continuous function, we are unable to determine the derivative at a point directly, but we can apply a discrete differentiation filter to approximate the derivative at a point.

$$G_x = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} * Image \approx \frac{\partial I}{\partial x} \tag{2.2}$$

$$G_y = \begin{bmatrix} -0.5 \\ 0 \\ 0.5 \end{bmatrix} * Image \approx \frac{\partial I}{\partial y} \tag{2.3}$$

where $*$ represents convolution.

G_x approximates the image gradient in the x-axis, while G_y approximates the image gradient in the y-axis.

2.1.2 Features

A ***feature***, in the context of computer graphics, is a generic term for some component of a visual input that can be tracked. Examples of features include edges, corners, and blobs. Features in image space typically represent a characteristic of the underlying 3-dimensional scene, like a blob of colour painted on a wall, or the corner of a brick. If a camera takes two images of such a real-world feature from two nearby positions, ideally the image representations of the feature will be similar. This allows a suitable feature detector to match the two views successfully.

Edges are generally defined to be locations in an image where the image intensity derivatives are of large magnitude; that is, where the intensity changes rapidly. This typically occurs at the edge of an object, or where there is a texture on an object, such as a painted sign. The most prominent edge detector is the Canny edge detector [9], which blurs the image, finds regions of high derivative, prunes the edge so that only strong edges are allowed, and attempts to connect up nearby sections of edge into a longer edge.

Edges are hard to match between frames, since they represent an area in the image that can deform significantly between frames, such as when one object occludes another behind it. In addition, edges only constrain the feature in one dimension, meaning that an edge can match

with any position along another edge. For these reasons they are typically not used in navigation systems.

Blobs are regions of an image that satisfy a particular condition, for example, pixels that are red, or brighter than their neighbours. A blob is detected when a region larger than a particular size satisfies the condition, perhaps representing a red object. The blobs are expanded to best fit the region, so there are typically few blobs in an image. Blobs are tracked from frame to frame by matching each blob to the nearest blob in the previous frame — since there are few, this is relatively reliable.

Blobs are typically not used for robot navigation systems, either, since they distort significantly with orientation, and large areas of colour are uncommon in many environments; instead, they are primarily used in special, controlled applications.

Corners are locations defined by the intersection of two edges, locations where the image intensity gradient is strong in two directions at once. Corners are found anywhere two edges cross, such as where two objects occlude a third, or are silhouetted against a background. There are several commonly-used corner detection algorithms in general use, the most common being FAST[10] and Harris–Stephens[3] corner detection.

Corners are generally good features to track because they typically do not distort significantly between successive frames. They are well-defined, and many algorithms exist for detecting image corners stably and matching them to distorted versions of themselves. Corners formed by the intersection of two occluding objects are unstable under changes of orientation, but corners formed by texture are stable. This work employs corners as its features, since corners provide more tracking information than edges or blobs in general environments.

2.2 Algorithms

This section investigates more deeply the two primary algorithms employed in corner detection, Harris–Stephens and FAST.

2.2.1 Harris–Stephens

The ***Harris–Stephens Corner Detector*** [3] was introduced in 1988 as a general-purpose detector, capable of detecting either corners or edges, depending on the value of tunable parameters. In essence, the detector finds regions where the image intensity derivatives in two directions are both strong, indicating that two edges cross.

Consider a small circular window positioned over an image. If we move the image underneath the window, the way in which the visible part of the image changes provides information about that local region. If the visible region is in a featureless area of the image, for instance a painted wall or sky, a small movement of the image will have minimal effect on the visible region. If the visible region shows an edge, a movement parallel to the edge will show minimal change, while movements in all other directions will show a significant change. If the visible area is a corner, however, every possible motion will show a significant change.

This window observation is formalised as a sum of squared image differences. For a given window, w, we integrate the squared difference between the image intensity function, I, and a version of itself, shifted by (δ_x, δ_y).

$$E(\delta_x, \delta_y) = \iint_w (I(u + \delta_x, v + \delta_y) - I(u, v))^2 \,\mathrm{d}u \,\mathrm{d}v \tag{2.4}$$

High values of E represent shifts with a large difference between the windowed regions, and low values represent shifts with a small difference. A featureless wall would be expected to have low E-values for all shifts, an edge would be expected to show low values for shifts parallel to the edge and high elsewhere, and a corner would have high E-values for all shifts.

Since we don't have access to the image intensity function, just a discretely-sampled version of it, we instead must use the discrete version of the above:

$$E(\delta_x, \delta_y) = \sum_{u,v \in w} (I[u + \delta_x, v + \delta_y] - I[u, v])^2 \tag{2.5}$$

The earlier Moravec detector [11] considers only shifts of $\delta_x, \delta_y \in \{-1, 0, 1\}$, and thus is only effectively able to detect corners and lines that are either axis-aligned or 45 deg from the axis. We want to consider all possible orientations of lines and corners, so this requires an anisotropic method of trialling shifts.

If we take $I[u - \delta_x, v - \delta_y]$ and perform a first-order Taylor series expansion around (u, v), we have

$$I[u - \delta_x, v - \delta_y] \approx I[u, v] + \frac{\partial I}{\partial x}(u, v)\delta_x + \frac{\partial I}{\partial y}\delta_y \tag{2.6}$$

and therefore

$$E(\delta_x, \delta_y) \approx \sum_{u,v \in w} (\frac{\partial I}{\partial x}(u, v)\delta_x + \frac{\partial I}{\partial y}(u, v)\delta_y)^2 \tag{2.7}$$

If we use a rectangular window, the result will not be anisotropic, as a small rotation of the image will cause a change in the pixels covered by the area. Thus we must choose a rotationally symmetrical window. A circular window is possible, although typically a Gaussian filter is used, since it smooths the output as well.

$$E(\delta_x, \delta_y) \approx \sum_{u,v} w(u, v)(\frac{\partial I}{\partial x}(u, v)\delta_x + \frac{\partial I}{\partial y}(u, v)\delta_y)^2 \tag{2.8}$$

where $w(u, v)$ represents a 2-dimensional Gaussian filter of some σ.

Expanding, we get

$$E(\delta_x, \delta_y) \approx \sum_{u,v} w(u,v) \left(\left(\frac{\partial I}{\partial x} \right)^2 \delta_x^2 + 2 \frac{\partial I}{\partial x} \frac{\partial I}{\partial y} \delta_x \delta_y + \left(\frac{\partial I}{\partial y} \right)^2 \delta_y^2 \right) \tag{2.9}$$

Using the result from section 2.1.1, we approximate the image derivatives with derivative filters:

$$\text{let } I_x = I * \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} \approx \frac{\partial I}{\partial x} \tag{2.10}$$

$$I_y = I * \begin{bmatrix} -0.5 \\ 0 \\ 0.5 \end{bmatrix} \approx \frac{\partial I}{\partial y} \tag{2.11}$$

$$\tag{2.12}$$

and further

$$\text{let } A = I_x^2 * w \tag{2.13}$$

$$B = I_y^2 * w \tag{2.14}$$

$$C = (I_x I_y) * w \tag{2.15}$$

Thus, for small shifts, the sum of squared errors can be expressed as

$$E(\delta_x, \delta_y) = A\delta_x^2 + 2C\delta_x\delta_y + B\delta_y^2 \tag{2.16}$$

This expression lends itself to a matrix representation

$$E(\delta_x, \delta_y) = \begin{bmatrix} \delta_x & \delta_y \end{bmatrix} \mathbf{A} \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} \tag{2.17}$$

$$\text{where } \mathbf{A} = \begin{bmatrix} A & C \\ C & B \end{bmatrix} \tag{2.18}$$

The matrix **A** describes the curvature of the autocorrelation function at the centre of the window, and its two eigenvectors and eigenvalues give the directions and magnitudes of the principal curvatures, respectively. Since the direction of curvature is unimportant to us, it is sufficient to examine the eigenvalues. If the two eigenvalues are both small, any small shift will have a small E, thus the point has high autocorrelation with its neighbourhood. This means that the point is in a region of the image with little gradient, and not a corner or edge.

If one eigenvalue is significantly larger than the other, it suggests a gradient in one direction. Shifts in this direction will result in a small E value, while shifts perpendicular to the direction will result in a large E value. This means that the point represents an edge.

If both eigenvalues are large, there are two strong gradients approximately normal to one another. Shifts in any direction will result in a large E, and the point represents a corner. This particular case is what we are seeking.

Calculating matrix eigenvalues is unfortunately a computationally expensive operation, and the Harris–Stephens corner detector introduces an algorithmic simplification, returning a Harris–Stephens ***corner score:***

$$score = \det(\mathbf{A}) - \kappa \operatorname{trace}(\mathbf{A})^2 \tag{2.19}$$

$$= AB - C^2 - \kappa(A + B)^2 \tag{2.20}$$

where κ is a free parameter. High scores indicate higher likelihood that a given point is a corner.

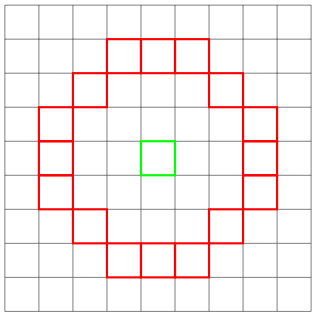


Figure 2.1: The FAST corner detector sample points

2.2.2 FAST Corners

Features from Accelerated Segment Test (FAST) is a computationally-cheap corner detector introduced in [10]. FAST examines a ring of pixels around a test point and reports a corner if a continuous run of pixels around the ring is brighter or darker than the centre pixel.

For each candidate pixel in the image, a Bresenham circle of radius 3 is constructed, comprising 16 points, as shown in figure 2.1. The pixel value at each of these points is compared to the centre, candidate pixel with a threshold. Given a threshold T, centre pixel intensity I_p and circle pixel intensity I_x, two separate comparisons are evaluated: $I_x < I_p - T$ and $I_x > I_p + T$. Thereafter, the two comparison sets are treated separately.

If a candidate pixel represents a corner, it is likely that at least 12 consecutive circle pixels of the 16 will be either darker or lighter than the centre pixel, as shown in figure 2.2. On a CPU, bit operations can be used to test for this quickly, and such pixels are reported as corners. However, FAST in 12-pixel mode is only able to detect sharp corners, and can fail to detect sharp 90-degree corners as well (fig. 2.3). To combat this, the test can be made more permissive, requiring between 9 and 11 consecutive pixels passing the test to determine a corner. FAST-9 is the most common of these, requiring 9 consecutive pixels to be lighter or darker than the centre pixel. FAST-9 is then vulnerable to spurious corners, such as shown in figure 2.4.

It is standard in FAST implementations to use FAST as an early-cull filter, passing candidate corners on to Harris–Stephens or another corner detector for refinement. FAST is a good first filter because it is so quick to calculate, meaning that the significantly more computationally-expensive Harris–Stephens detector only runs on a subset of the input pixels. This approach can lead to the situation where a pixel that would have returned a high Harris–Stephens corner score is discarded at the FAST stage, and thus is not represented in the output, but with tuning of the

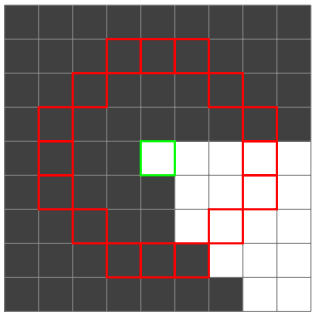


Figure 2.2: FAST-12 identifies this corner successfully, since 13 consecutive pixels are darker than centre pixel

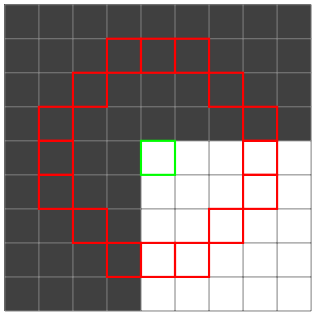


Figure 2.3: FAST-12 can fail to detect sharp 90-degree corners, since only 11 pixels are darker than the centre pixel

threshold value, this situation can be made rare, and losing a small number of features does not typically affect the efficacy of later processing steps.

FAST itself does not return a corner score for pixels, just a binary "is-a-corner" value. In some applications this is sufficient, but its output is unsuitable for non-maximum suppression. Since non-maximum suppression is critical for increasing the signal-to-noise ratio of detected features, in most implementations, FAST is used as an early cull, and corner candidates are then checked with Harris–Stephens afterwards. Since FAST is so much simpler than Harris–Stephens, its runtime is much shorter, and Harris–Stephens is only used to check possible corners reported by FAST, reducing total feature detection time.

2.2.3 Non-Maximum Suppression

Both Harris–Stephens and FAST will often return a cluster of corner pixels in the neighbourhood of a corner, since pixels slightly offset from the point of a corner will often have high corner scores or pass the FAST test, respectively. Figure 2.5 shows the corner scores for pixels in the vicinity of a corner point, demonstrating a peak in the corner score dimension. One central

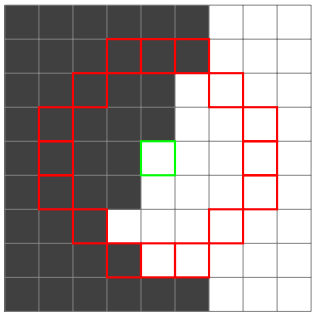
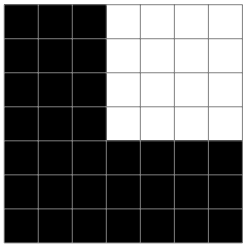
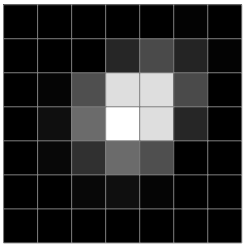


Figure 2.4: FAST-9 identifies this pixel as being a corner, despite it being some distance from the actual corner point, since 9 consecutive pixels are darker than the centre



(a) Input image



(b) Corresponding Harris–Stephens corner scores (normalised)

Figure 2.5: Corner scores emitted by the Harris–Stephens detector. White indicates higher values. The corner scores on the right correspond to the pixels on the left. The corner scores were generated using a 7×7 Gaussian smoothing window. [2] ©IEEE 2017

pixel has the strongest score, but others around it also have high responses.

Since this peak represents only one corner in the input image, it would be nonsensical to treat each pixel in the peak as an independent corner, and could jeopardise later matching (in particular when prominence filtering is enabled, as described in section 3.6.1). To that end, we want to select only the pixel with the maximal score, and suppress all others.

Non-maximum suppression is an algorithm for this. For each candidate corner, the corner scores of the neighbouring pixels in a square window are examined. If any of those scores is higher than our candidate's, it is excluded from the output stream. This ensures that only local maxima in the corner score surface are selected.

2.3 Hardware Primitives

2.3.1 Sliding Window

This work makes extensive use of a 2-dimensional ***Sliding Window*** structure, which allows easy and fast access to pixel values in a small rectangular window. When pixels from an image are fed into the sliding window one at a time, in a raster scan, the rectangular window will progressively iterate over every position in the image, providing access to the image pixels covered by the window.

The sliding window structure is a common choice in computer vision applications on FPGA, begin used as a primary component in [12], [13] and [14].

As shown in figure 2.6, each row of the window is constructed as a shift register, with one 8-bit register for each pixel in the row. A series of line buffers (see section 2.3.2), implemented in RAM, allow the input to each row of the window to be delayed by the width of the image (`image_width`), and so the start of each row's shift chain is fed with the pixels forming a column of the input image. This ensures that all of the rows of the sliding window are synchronised, and that a given x offset into the window represents the same image x-coordinate in all the rows.

The dimensions of the shift register array, `WINDOW_WIDTH` and `WINDOW_HEIGHT`, are specified at design time, as described in appendix A. The parameter `image_width` is configurable at runtime. The maximum image width, `MAX_IMAGE_WIDTH`, must be specified at design time, as this determines the amount of memory to be allocated. At the beginning of each frame, `image_width` is programmed with the width of the image, ensuring that the linebuffers align correctly.

The sliding window does buffer large numbers of input words, in effect causing a pipeline delay. The delay is due to the necessity of waiting for future pixels before the neighbourhood of a given pixel is available. This delay is proportional to the width of the image and the height of the window:

$$delay = \left\lfloor \frac{\texttt{WINDOW\_HEIGHT}}{2} \right\rfloor \times \texttt{image\_width} + \left\lceil \frac{\texttt{WINDOW\_WIDTH}}{2} \right\rceil \text{ cycles} \tag{2.21}$$

The sliding window offers a number of desirable traits. It keeps its own state internally, and all it requires is a single pixel to be input each clock cycle. In addition, each pixel in the

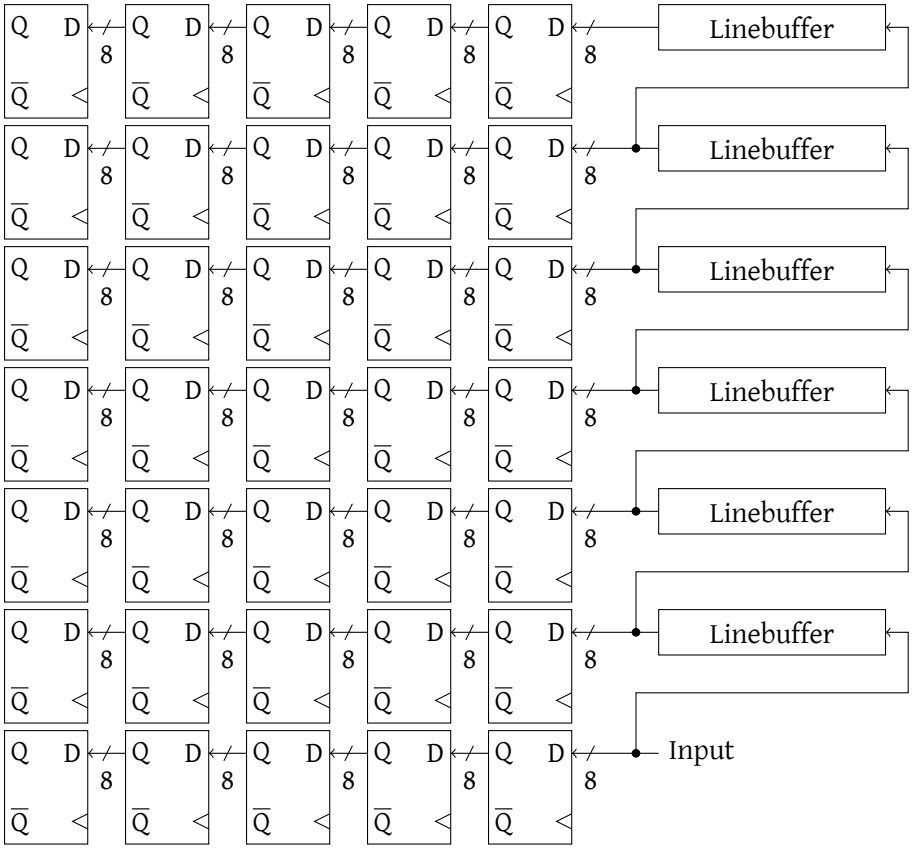


Figure 2.6: A block diagram of an 8-bit 5×7 sliding window. The diagram is arranged so that the registers represent pixels in the order they appear in the image, as a rectangular window of the source. The top-left register represents the top-left pixel of the window, the bottom-right register represents the bottom-right pixel of the window. This means that input flows into the window right to left, bottom to top, opposite to the normal schematic convention.

input image only needs to be read once. Access to specific cells of the sliding window is very simple. The output from each word register of the shift register is routed into the next in the sequence. Some of the output values will be fanned out into other logic that performs a function using the window as input. For instance, a gradient function could be implemented by taking the difference between two adjacent cells in a sliding window's shift register array. Since the values come directly out of registers, this means that the cells can be accessed with no latency, and requiring minimal additional routing.

The structure does have a number of disadvantages. In addition to the shift register chains, it is necessary to store multiple rows of the image in row buffers. The total memory consumption of the row buffers (in bytes) is

$$(\texttt{window\_height} - 1) \times \texttt{MAX\_IMAGE\_WIDTH} \tag{2.22}$$

MAX_IMAGE_WIDTH is typically a power of two, since it represents a RAM address. The number of linebuffers needed is one less than the height of the window. It is not necessary to buffer the input to the first (bottom) row of the sliding window, since that is fed directly from the input, as shown in figure 2.6.

The other primary disadvantage of the sliding window is that it does not directly support random access to the pixels within. Since the registers are implemented using flip flops, there is no addressing logic implicitly provided. External random-access circuitry, in the form of multiplexers in the x- and y-axes, would provide random access at the cost of large logic usage and latency, but typically block RAM offers a better solution in these cases, as described in section 3.4.2.

If both random access and fast fixed access are required simultaneously, a logic-efficient implementation is one sliding window and one RAM-backed window, handling each access separately.

2.3.2 Configurable Linebuffer

In between each pair of rows of the sliding window is positioned a linebuffer. The purpose of this module is to form a shift chain, the length of `image_width`, so that each row of pixels is delayed by the correct amount to align with the row below. In addition, in order to support images of

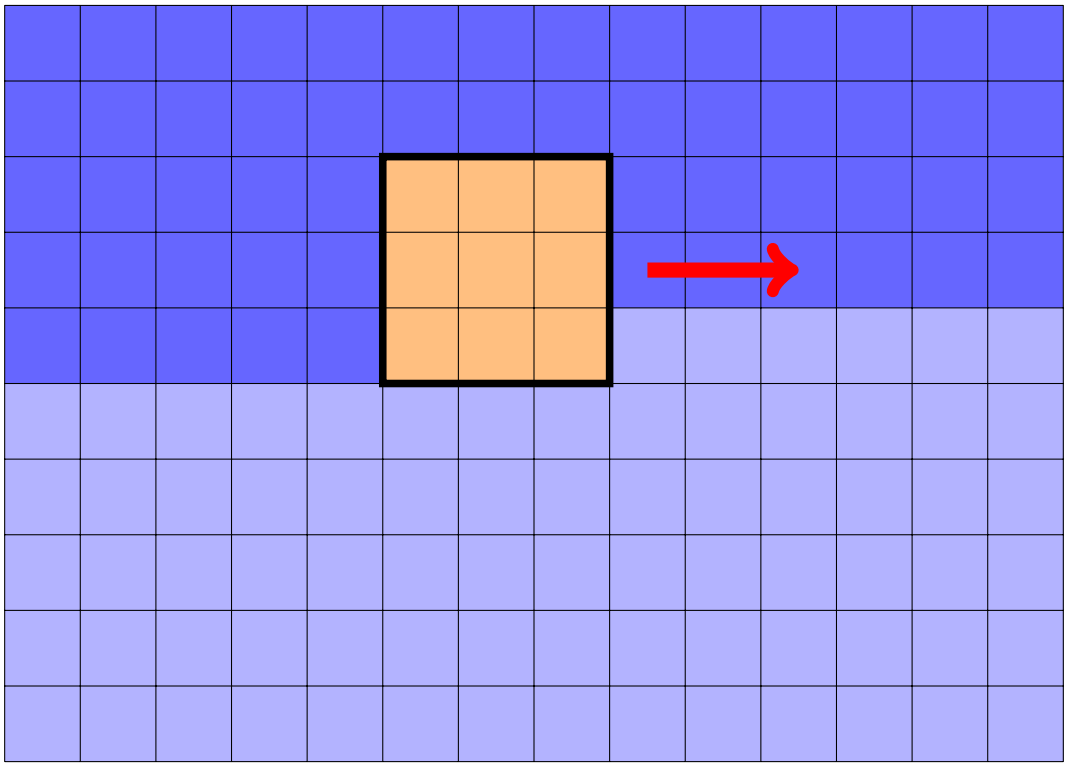


Figure 2.7: The coverage pattern of a sliding window across a window. Light blue represents pixels not yet covered, dark blue represents pixels that have been covered by a window, and orange shows the current contents of the window. The window scans left to right, top to bottom.

variable widths, it is necessary to have `image_width` be configurable to suit the current frame.

The linebuffer could be implemented using logic elements, but because this would use a prodigious amount of logic, particularly for large images, and because it would complicate runtime reconfiguration, this system instead uses a block RAM (BRAM)-based linebuffer. A dual-ported BRAM is used, with one read port and one write port. The length of the RAM is dictated by the maximum supported image width, `MAX_IMAGE_WIDTH`. Both ports access the same address, and the address counter register increments every time a value is written. The address counter register will reset to 0 when it is incremented to reach `image_width`. This ensures that the delay between a value being written and being read is exactly `image_width` write operations.

Upon module reset, the `image_width` is loaded from an external source.

2.3.3 Margins

All of the image processing algorithms used in this work operate on a rectangular window centred at a particular point, with some input pixels to the left, right, above and below the centre.

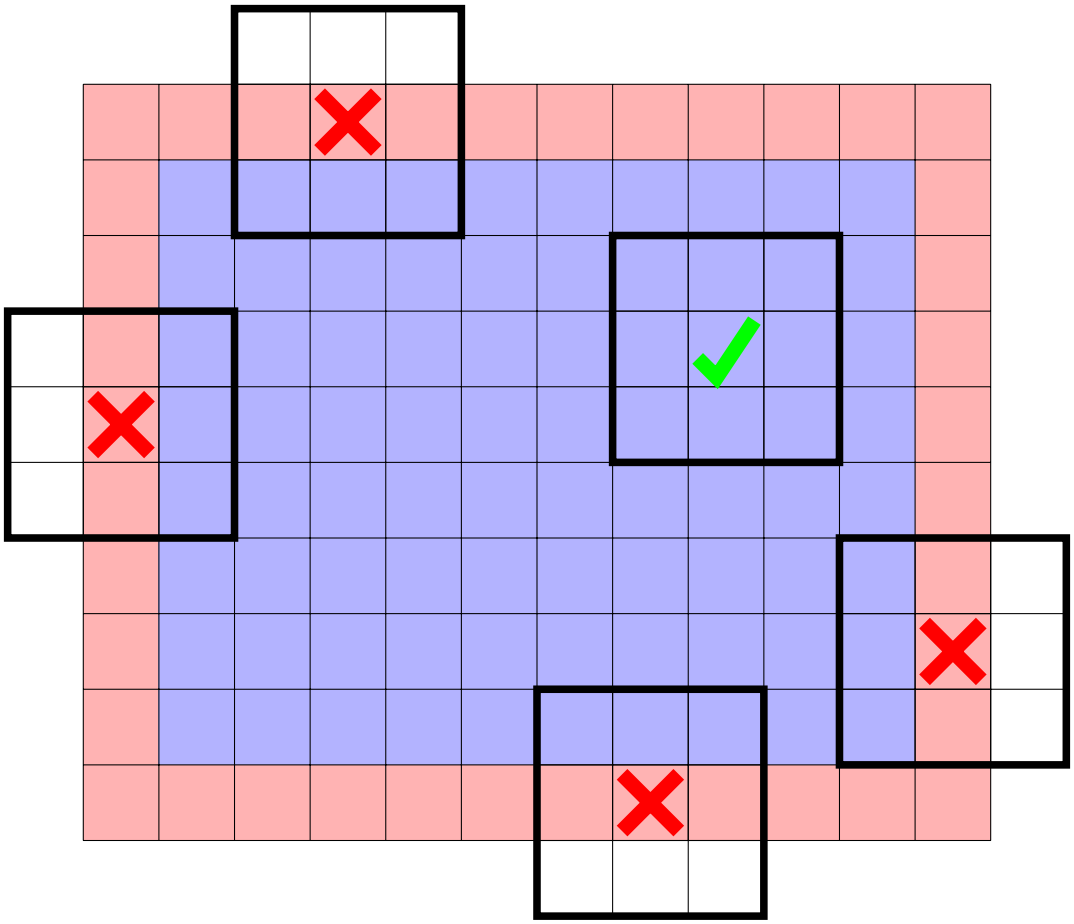


Figure 2.8: At the image margins, insufficient image data is available

This means that, for pixels near the edges of the input image, insufficient input data is available for calculation.

There are a number of different approaches for dealing with this situation mathematically. Pixel accesses outside of the window can be wrapped around to the other side, or mirrored back onto the same side. Wrapping is, however, unsuitable for most images, since one side of the image is typically unrelated to the other side.

Method	Virtual coordinate	Effective coordinate
Wrapping	(-2,5)	(30,5)
Wrapping	(5,-2)	(5,30)
Mirroring	(-2,5)	(2,5)
Mirroring	(5,-2)	(5,2)
Sliding Window Naive	(-2,5)	(30,4)
Sliding Window Naive	(5,-2)	n/a

Table 2.1: Effective coordinates for virtual coordinates on a 32×32 window

Mirroring and wrapping are both straightforward to implement using random-access hard-

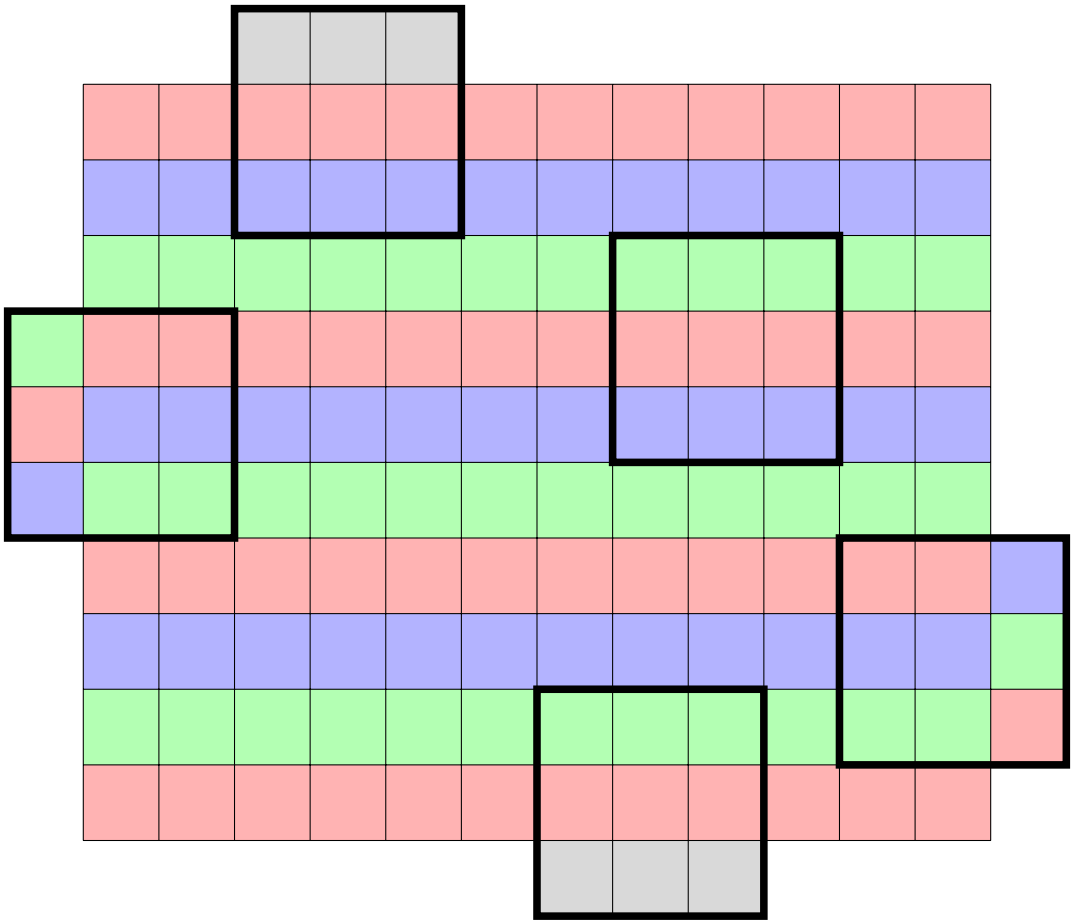


Figure 2.9: The contents of the sliding window at different positions in the margins of an image. Windows projecting off the edge of the image contain image data from previous rows (in the case of the left edge) and future rows (in the case of the right edge).

ware. Each coordinate access is checked before the read is performed, and if necessary, is modified so that it is within the image bounds. In a streaming, sliding window-based architecture, such transformations require more complicated implementations, involving extra hardware.

Unmodified Sliding Window

Along the top and bottom edges of the image, a sliding window will contain garbage data for the part of the window outside the bounds of the image. Along the sides of the image, some image data from the opposite side of the image will be wrapped, but with a one-line offset, as shown in figure 2.9.

Wrapping

The sliding window can be modified to perform wrapping in the x axis relatively easily. Almost all the required pixels are already present in the sliding window, just offset by one row. In order to access them, the y-coordinate must simply be increased by 1. These slightly increases the complexity of the routing, since two signals must be routed to the relevant logic, with a multiplexer to select between them.

Wrapped accesses in the bottom row of the window are more complicated, since the pixel data is not yet present in the window — it has not yet arrived, and is still a full row away. This can be worked around by making the window one row higher than otherwise needed, adding another row of delay. This will ensure that the required data is present when needed.

Wrapping in the y-axis, however, is essentially impossible without buffering an entire frame, since the wrapped pixel data is not present in the window at all. For windows along the top of the image, wrapped pixel accesses would be at the bottom of the image, which has not yet entered the window and will not for a long time. Along the bottom of the image, the accesses refer to image data that exited the window and the system long ago, and is no longer available. While it would be possible to buffer the pixel data in this case, the buffer would have to be the full size of the image, which is highly impractical (see section 3.4.5).

Mirroring

Mirroring can be implemented relatively straightforwardly on a sliding window, since the needed pixels are always present in the window. The complexity comes from the routing logic required to access the pixels. Since the mirrored coordinate depends on how far out of bounds the access is, there are now many potential window coordinates which contain the desired pixel. This greatly increases the amount of routing needed to synthesise the design, and could lead to delays because of the increased fan-out.

Coordinate mirroring also has the disadvantage that false information is being created and treated as true input, where image samples are being treated as if they are from elsewhere. This can have impacts on the quality of the processing.

Dropping

The simplest approach, and the one employed in this work, is to regard the margin region of the image as being invalid. The size of the margin is defined by the size of the window: for a window of size $m \times n$, the x-axis margin is $\frac{m-1}{2}$ and the y-axis margin is $\frac{n-1}{2}$. Any window centred at a point within the margin is invalid, since at least part of the window is invalid. Any window outside the margin is guaranteed to be entirely valid.

This does mean that no features can be detected in a portion of the image around the border. The larger the image, the less significant this border region is, but the region is a fixed width regardless of pyramid level. Since higher-level images are smaller, a higher proportion of the image pixels are lost.

The input to the Harris–Stephens corner detector is a 3×3 window of image pixels, which would suggest a margin of only size 1. However, within the detector itself, further windows are used. As described in section 2.5.1, the implemented Harris–Stephens detector has three sliding windows, with sizes

$$(3 \times 3) \rightarrow (7 \times 7) \rightarrow (3 \times 3)$$

The margins for chained windows are additive, thus the total margin is 5, for an effective window size for the Harris–Stephens detector of 11×11.

The window used in ORB feature extraction, however, is 37×37, as described in section 3.4.5. Since the two operations (Harris–Stephens corners and ORB extraction) are functionally independent, the combined margin of the two is not the sum, but rather the maximum of the two, namely 18.

Level	Dimensions	Usable dimensions	Usable
0	640×480	604×444	87.30%
1	512×384	476×348	84.25%
2	409×307	373×271	80.50%
3	320×240	284×204	75.44%
4	256×192	220×156	69.82%
5	204×153	168×117	62.98%
6	160×120	124×84	54.25%
7	128×96	92×60	44.92%
8	102×76	66×40	34.06%

Table 2.2: Usable image area with pyramid level

Figure 2.2 shows the usable image area for a selection of image sizes. Since the window margin for all image sizes is 18 pixels, a strip 18 pixels wide is unusable along the perimeter of all the images. In the case of small images, this strip represents more than half of the total image area.

The image sizes are derived from the dimensions of the levels of an image pyramid rooted at 640×480, as described in 4.1.1. Note that the usable area of pyramid levels as a percentage falls with increasing levels of the pyramid.

2.4 Previous Work

2.4.1 Harris–Stephens

Several FPGA implementations of the Harris–Stephens feature detector have been published, all employing a sliding window structure, as described in section 2.3.1.

Hsiao, Lu and Fu [13] (2010) employ a chain of three separate sliding windows, with each window in the chain handling a separate part of the algorithm. Each stage of the Harris–Stephens algorithm executes independently of the others, allowing effective pipelining. While the first window in the chain is processing newly-arrived pixels, the later windows are processing previously-computed outputs.

The implementation uses a slightly unconventional formulation of Harris–Stephens, performing a Gaussian blur before taking the image derivatives. Consequently, a 9×9 window is used for extracting derivatives, a 7×7 window is used to perform a Gaussian filter on the derivative products to calculate the structure tensor, and a final 3×3 window is used for non-maximum suppression. No details are given as to the bit width used for the intermediate calculations. Hsiao, Lu and Fu report a performance of 46 fps for 640×480 images, which is equivalent to 14 MPix/s, on a Cyclone II FPGA. No details are given for the latency, but it must be at least $8 \times$ `image_width`, given the sizes of the sliding windows.

Possa, Mahmoudi, Harb, Valderrama and Manneback [12] (2014) present a similar architecture to [13], which has much higher performance. A more conventional approach is taken to calculating derivatives, employing a standard 3×1 convolutional derivative filter, 5×5 Gaussian filtering of the structure tensor elements, and a large $9\times$ non-maximum suppression window. Interestingly, the Harris–Stephens score is only calculated with 8 bits of precision, saturating large values to 255. This is to minimise the storage space required for the 9×9 sliding window's linebuffers, but it has the problem of creating regions of the image with equal corner scores (all 255). This is mitigated by inserting an additional 5×5 Gaussian low-pass filter, at the cost of added latency and storage.

Additionally, instead of invalidating windows in the margins of the image, the implementation uses the "mirroring" strategy described in section 2.3.3. This is a problematic choice, since the "mirroring" strategy creates information that is not present in the original image. All other

implementations adopted the "drop" strategy, since it does not introduce any additional information.

The authors report an operating speed of 232 MPix/s on an Arria V FPGA (the same as used in this work). Latency is $9 \times \texttt{image\_width} + 53$ cycles. Additionally, they measure power consumption at 6.7 mJ for a 1024×1024 image.

Amaricai, Gavriliu and Boncalo [15] (2014) present a low block-RAM-usage variant on this approach, specifically for situations where on-chip RAM is limited, but logic elements are not. The work makes the observation that a large part of the memory cost of the design in [13] is due to persisting the calculated Harris matrices and Harris–Stephens scores between rows. These values are stored in a block RAM, which must be at least as wide as the image.

The design presented uses "tall" windows to calculate multiple values simultaneously, and discards these values after use. The tall windows contain the information for multiple overlapping input windows, and these are computed in parallel. "Taller" sliding windows are used so that intermediate values can be calculated for several rows at once. The values are not persisted between image rows, and must be recalculated on the next pass.

To realise this, only one sliding window is employed, windowing the input pixels. Instead of storing previously-calculated intermediate values—such as derivative products and Harris–Stephens scores—in a sliding window, these values are instead recalculated every row. Intermediate results are buffered in a shift register for that row, so some reuse occurs, but significantly less than either of the above approaches.

The one sliding window is a 3×7 pixel window on the input. This constitutes 5 overlapping 3×3 windows, each of which provides the input to a derivative calculation module. The output derivatives are multiplied to yield the derivative products, and thus we have a 1×5 vector of 2×2 Harris–Stephens matrices. This is used as the input to a 3×5 shift register, constituting 3 overlapping 3×3 windows, which are the inputs for 3 Gaussian filters. The outputs from the Gaussian filters are a 1×3 vector of structure tensors, which are the inputs for 3 Harris–Stephens score calculation modules. Finally, the 1×3 vector of corner scores provides input for a 3×3 shift register, and non-maximum suppression is performed on this.

In effect, each Harris–Stephens matrix is calculated five times, each smoothed structure tensor three times, and each corner score three times. This imposes no extra latency over the alternative methods, but does impose a cost in logic and in power consumption. The authors

report a performance of 225 MPix/s on a Virtex-5 FPGA, but no power consumption figures.

Latencies are given for many of the modules, but not the system total. Using the provided latencies and estimates based on the implementations in this work, total system latency is estimated as $3 \times$ `image_width` $+ 12$ cycles. This figure is significantly lower than those for the other two implementations primarily because the windows are so much smaller. A 3×3 window introduces only 1 row of latency, and all three "effective" windows are 3×3.

2.5 Implementation

This section describes the hardware implementation of a feature detector, comprising a Harris–Stephens corner detector (see section 2.2.1) and non-maximum suppression (see section 2.2.3). The mathematics of the implementation, as well as latency and precision, are discussed, and the performance is compared to other existing implementations.

This work does not use the FAST feature detector. In CPU implementations, FAST is almost never used without Harris–Stephens as well, since FAST does not return a corner strength score. It is instead used as an optimisation, because it reduces the computational load on the Harris–Stephens detector by filtering out points that are definitely not corners.

This hardware feature detection implementation is highly-pipelined, and guarantees a 1 pixel per clock throughput, since there are no pipeline stalls. The logic for Harris–Stephens corner detection is completely independent of all other stages, and thus there is no advantage to early exit, if it is determined that a pixel could not possibly represent a corner. The Harris–Stephens logic would still compute a result, which would then be ignored. This contrasts with a CPU implementation, where the same processor is used for calculating Harris–Stephens scores and doing other tasks, such as feature matching.

It must be noted that the combination of FAST and Harris–Stephens does not have identical output to just Harris–Stephens alone. It is possible for pixels to be arranged such that FAST will not detect a corner, but Harris–Stephens will report a strong corner. In such cases, Harris–Stephens alone will detect more corners than the combination. This is not considered to pose a problem.

The input to the detector is a stream of pixels, one per clock cycle. The output is a single-bit signal indicating whether the corresponding pixel coordinate is determined to be a corner or not. The output signal is delayed relative to the input, since later pixel values are required to determine whether a coordinate is a corner or not. See figure 2.11 for per-module latency details.

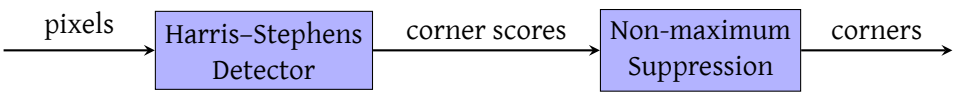


Figure 2.10: Overview of the Corner Detection system

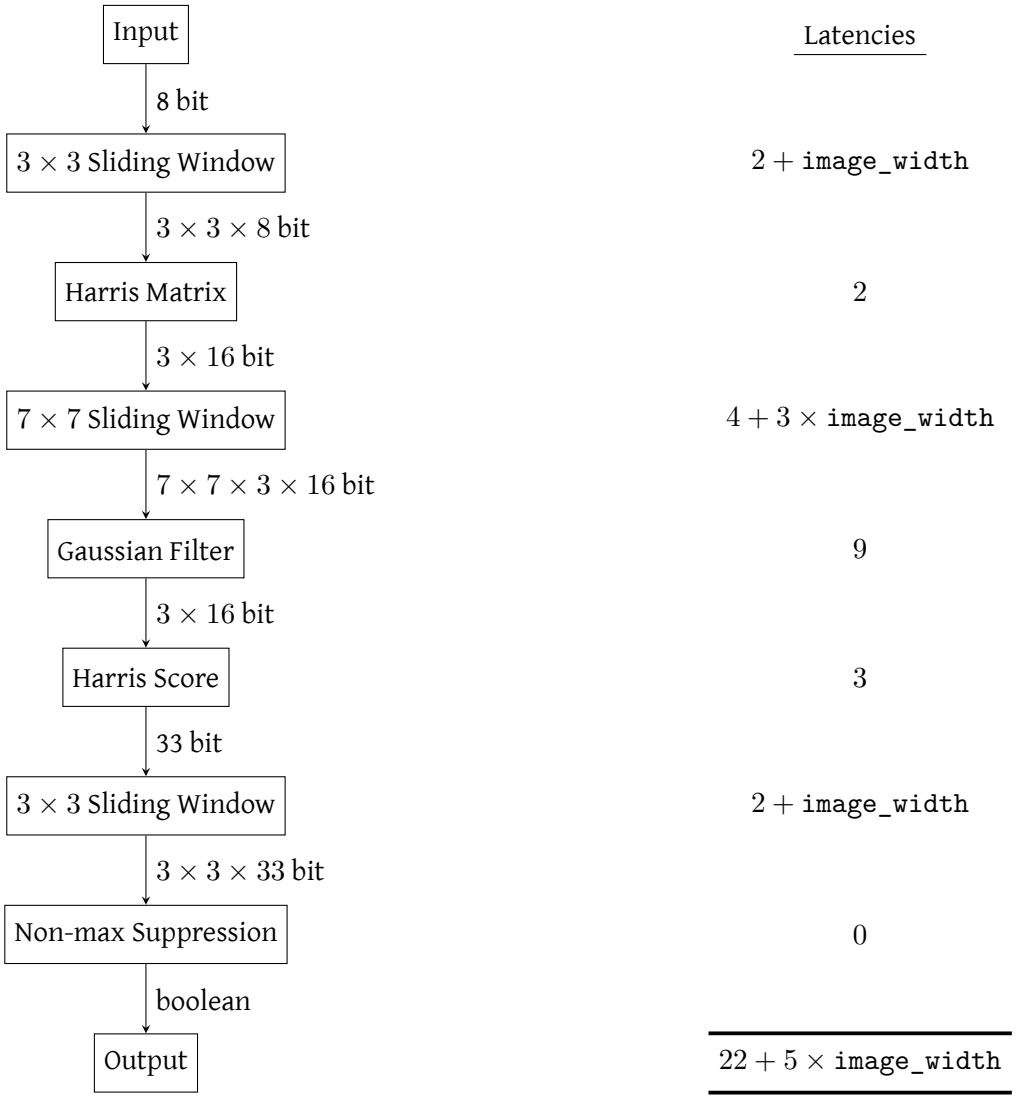


Figure 2.11: Data flow and latency in the Harris-Stephens corner detector

2.5.1 Harris–Stephens Detector

Introduction

This section describes a hardware implementation of the Harris–Stephens feature detector. The input to the module is a stream of pixels, and the output is a ***corner score*** for each pixel coordinate.

This implementation is similar to those in [12], [13], since they are architectures that represent efficient use of logic and yield 1 pixel/clock throughput. The approach of [15] represents a different tradeoff, for situations where logic is plentiful, but memory is constrained. The FPGA used in this work, an Arria V, has plentiful memory, so this is not considered necessary.

Gradient calculation

The input pixels flow into a 3×3 sliding window. This sliding window contains enough information to approximate the image gradients using the Sobel operator[16]. This operator is used instead of a simpler derivative filter because it combines a smoothing operation with the derivative calculation, and draws gradient information from the local neighbourhood of the point. The Sobel operator convolves the input image ($\mathbf{A}$) with two 3×3 kernels, one for the gradient in the x-axis, one for the gradient in y:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathbf{A}, G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathbf{A}$$

This convolution is performed in hardware with shifts, adds and subtracts.

Harris matrix calculation

Next a "Harris Matrix" is constructed, which is the outer product of a vector of the two gradient components with itself:

$$\mathbf{H} = \begin{bmatrix} G_x \\ G_y \end{bmatrix} \begin{bmatrix} G_x & G_y \end{bmatrix} = \begin{bmatrix} G_x^2 & G_x G_y \\ G_x G_y & G_y^2 \end{bmatrix}$$

Since the $\mathbf{H}_{12}$ and $\mathbf{H}_{21}$ components are identical, we can save a multiplication, and so only three multipliers are needed. The implementation of these first two steps can be found in *HarrisMatrixPipelined.sv*.

Gaussian filtering

The next step is to smooth the gradients using a Gaussian filter. The standard deviation σ of the filter is a free parameter of the detector, and this work uses a sixth-order binomial filter, with a kernel of size 7:

$$\begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix}$$

Higher- and lower-order Gaussian filters were briefly tested, but little effect on the output was observed, so this parameter was not investigated in depth. The size of the window has a significant impact on the resource consumption, however, since each cell of the window requires three wide multipliers and each additional row requires an additional row of Harris matrix coefficients be stored in memory. This invites further investigation of its effects.

The standard deviation of a one-dimensional binomial distribution is given by $\sigma = \frac{\sqrt{n}}{2}$, where n is the order of the filter. When two separable one dimensional filters are convolved with one another orthogonally, as here, the standard deviation of the combined filter in each axis is unaffected. Since both filters have order 6, the combined filter has a standard deviation of $\frac{\sqrt{6}}{2}$ in both the x- and y-axes.

This filter is applied individually to each of the components of **H**, yielding a matrix called the ***structure tensor***, **A**, which represents the smoothed Harris matrix. Each multiplication in this convolution has one input constant, so they are implemented as shifts and adds, instead of using DSP multipliers, then the products are summed using a pipelined adder tree.

Corner score calculation

The Harris–Stephens corner score is derived from the components of **A**:

$$score = \det(\mathbf{A}) - \kappa \operatorname{trace}(\mathbf{A})^2 \tag{2.23}$$

In expanded form:

$$score = \mathbf{A}_{11}\mathbf{A}_{22} - \mathbf{A}_{12}\mathbf{A}_{21} - \kappa(\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{A}_{11} + \mathbf{A}_{22}) \tag{2.24}$$

Constant κ should be a small constant. This work chooses $\frac{1}{16}$, a power of two, since it allows the multiplication to be replaced with a shift, enabling calculation of the score with only three multipliers. The effect of modifying κ has not yet been investigated in detail, and would be a good avenue of further research.

The SystemVerilog implementation of the smoothing and score calculation is in *HarrisCornersPipelined.sv*.

Bit widths

The selection of bit widths used in various intermediate calculations of the Harris–Stephens feature detector is a significant subproblem. In general, the more bits of precision, the better the detector operates, since it is able to detect finer differences in corner scores. However, memory usage is proportional to the number of bits used, since a high-precision value requires a wider memory.

[12] attempts to sidestep this issue by using saturating arithmetic, which clamps overflow values to the maximum and underflow values to the minimum. This nonlinear transformation destroys information, however, and so this work does not use such an approach.

For simplicity and efficiency, this work uses solely integers. Some values, such as the input pixels, are unsigned integers, while others, such as the structure tensor and the corner scores, are two's complement signed integers.

In order to reduce memory requirements, this work reduces the precision of some intermediate values. There are sliding windows for the input pixels, for the Harris matrices, and for the Harris–Stephens scores, so these are the most effective values to target. The value is calculated to full precision internally, and then truncated before entering the sliding window. With the exception of thresholding, all the subcomponents of Harris–Stephens operate on relative magnitude, so truncating values has no effect on the algorithm result, aside from precision issues. These are explored in section 5.3.

The input pixels are 8-bit unsigned integers, representing greyscale values, with 0 being black

and 255 white.

Value	Range (8-bit input)	Bits needed	Bits (k-bit input)
Input pixels	$[0, 255]$	8 (unsigned)	k
Sobel response	$[-1020, 1020]$	11 (signed)	$k + 2$
Harris matrix terms	$[-1040400, 1040400]$	21 (signed)	$2(k + 2)$

Table 2.3: Bitwidths needed for full precision calculating Harris matrix terms

Table 2.3 shows the precision required to calculate Harris matrix terms to full precision. The maximum Sobel response is approximately 4 times the maximum pixel input, and the minimum is -4 times the minimum value. The Harris matrix terms are products of the Sobel responses. The diagonal elements are guaranteed to be non-negative, since they are squares, whereas the off-diagonal terms (which are identical) can be negative or positive.

As discussed in section 5.3, 21 bits of precision for the Harris matrix terms is more than required, and so the values are truncated before passing into the sliding window.

Value	Range (8-bit input)	Bits needed	Bits (k-bit input)
Input Harris matrix terms	$[-128, 127]$	8 (signed)	k
Sum of weighted responses	$[-524288, 520192]$	20 (signed)	$k + 12$
Re-normalised structure tensor	$[-128, 127]$	8 (signed)	k
Harris–Stephens score	$[-17408, 12097]$	16 (signed)	$2k$

Table 2.4: Bitwidths needed for full precision calculating Harris–Stephens score (7×7 Gaussian filtering)

Table 2.4 shows the precision required to calculate the Harris–Stephens score for a point to full precision. The weights in a 7×7 binomial Gaussian filter sum to 2048 (2^{12}), so the sum of the weighted components requires 12 bits more than the input matrix terms. In effect, each weight is a fraction with denominator 256, and so the weighted sum should be re-normalised afterwards by right shifting 12 bits.

The range of the Harris–Stephens score is complex to evaluate. For a structure tensor

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} \tag{2.25}$$

the maximal value is obtained when $a = c$ and $b = 0$. The minimal value is obtained when $a = 0$ and $b = c = -128$, or equivalently $a = b = -128$ and $c = 0$. This has the effect of leaving the range unbalanced, with a larger negative range than positive.

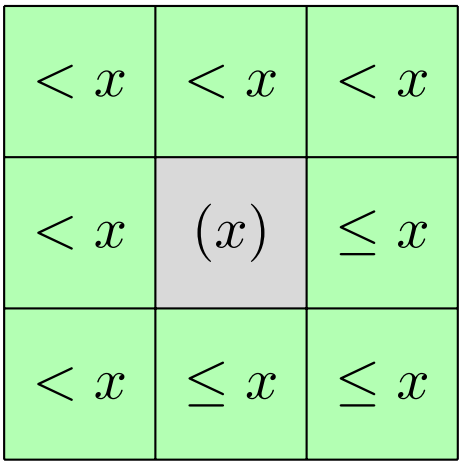


Figure 2.12: The tests performed for non-maximum suppression. x is the value of the centre pixel.

2.5.2 Non-Maximum Suppression

Non-maximum suppression is implemented as a post-processing stage following Harris score calculation. The scores from the pipelined Harris–Stephens corner detector pass into a 3×3 sliding window. For each position of the window, the centre score is compared against a threshold, which defines the minimum response required to detect a corner.

If a given corner candidate has a higher corner score than the threshold, it must then be compared against its neighbours to determine whether it is the local maximum score. This is done by comparing the score against the scores in the eight neighbouring pixels, which are present in the window. To better break ties, a corner candidate is reported as being the maximum if its score is *greater than or equal to* all its neighbours to the left and above, and if its score is *greater than* all its remaining neighbours. This ensures that one point from two adjacent pixels with the same score is selected and reported as being a corner.

If a corner candidate passes all these tests, it is determined to be a corner. The output from the module is a binary value indicating whether the current pixel is a corner or not. This signal is then used to trigger ORB modules in the second part of the system.

2.5.3 Performance

There are two main aspects of performance for a feature detector, timing and accuracy.

Timing

Timing has two major aspects, throughput and latency.

Throughput is the rate at which input can be processed. This system, and the three other examples presented in section 2.4.1, are all able to sustain a throughput of 1 pixel per clock cycle, meaning that the input is handled as quickly as it is received, since there are no possibilities for pipeline stalls. Thus the main factors affecting actual throughput are the clock speed and the rate at which pixel input can be supplied to the FPGA. The entire system, as implemented, attains a maximum clock speed of approximately 150 MHz, which is lower than [12], [15] at around 230 MHz, but neither of these systems included a feature extractor, and so the comparison is not entirely fair.

Latency is the delay caused by the pipelining of the algorithm. This work defines module latency as the delay between when a particular pixel enters the detector and when the detector reports its corner score. There is necessarily a delay, since the detection of the corner relies on information that will arrive after the pixel. Pixels from later rows of the image contribute information to the detector, and thus earlier information must be buffered.

In this system, and in all the others examined, the latency is dependent on the width of the image, since multiple rows must be buffered. In effect, each corner score is only delayed by a small number of clock cycles after the last, critical pixel is received, which provides the last piece of information required to calculate the output value.

The latency of each of the implementations is primarily dependent on the sizes of its sliding windows. This system has a total latency of $5 \times \texttt{image\_width} + 22$ cycles, with the window sizes configured as shown in figure 2.11. [15], with only 3×3 windows, has the lowest latency of any of the systems considered, at only $3 \times \texttt{image\_width} + 12$ cycles. [12], with its large windows, has the highest latency of all considered, at $9 \times \texttt{image\_width} + 53$ cycles.

It is important to realise, however, that the cycle time in all these systems is very fast, and so the latencies are on the order of tens of microseconds. For an image 1920 pixels wide (eg 1080p video), this Harris–Stephens detector implemented in this work has a latency of 64 µs.

Frame Latency is a combination of throughput and pixel latency. It measures the time between a when a frame begins being sent to the detector and when the detection results are available, and is a useful high-level abstraction of how long it takes to process an image. The

frame latency is given by:

$$Frame\,latency = Latency + \frac{Pixels}{Throughput} \tag{2.26}$$

For a standard 1080p video frame (1920×1080), the system implemented in this work will take 2083222 clock cycles, or 13.89 ms at 150 MHz.

Accuracy

Accuracy is the second factor in feature detection performance. Accuracy is very hard to quantify, since it is all but impossible to define a "ground truth" for the process of feature detection. The ground truth for an image would consist of a set of coordinates that are corners, and algorithms could be tested to ensure they return exactly this set when the same image is input.

Unfortunately, for all but the simplest geometric images, it is impossible to construct a ground truth, since real images have texture and gradients. These ensure that any two feature detection algorithms are almost guaranteed to detect the same features, and mean that comparing two different implementations of the same feature detector is difficult.

As described in chapter 5.3, this work instead uses simple corner counts at a given threshold to evaluate the performance of particular internal parameters.

2.6 Conclusion

This chapter has provided an introduction to feature detection, a selection of algorithms for corner detection, a history of hardware implementations of Harris–Stephens, and details of the hardware implemented in this work. Harris–Stephens has been selected as the sole corner detector, instead of the more common combination of FAST and Harris–Stephens.

The reason for this is that the FAST detector only provides an early cull of possible corner candidates, it does not actually return the necessary corner scores. FAST will reduce the computational load on a Harris–Stephens detector implemented on a CPU, but in this hardware implementation, the Harris–Stephens module will run regardless of any FAST preemption, and thus FAST is unnecessary. For more details, see section 2.5.

The architecture presented here is very similar to those of previous works, specifically Possa, Mahmoudi, Harb, Valderrama and Manneback [12] and Hsiao, Lu and Fu [13]. Both these papers, and this work, describe systems built from sliding windows, with the operations being performed on these windows. The throughput of all of these is identical on a per-cycle basis, and thus depends entirely on the clock speed of the implementation. This work has a significantly higher clock speed than [13] (150 MHz vs. 14 MHz), but a lower clock speed than [12] (232 MHz). The latency and hardware requirements for the three designs are dependent on the window sizes selected, and on the bit widths of the windows. For the same parameters, each design should have essentially identical latency and requirements.

The paper from Amaricai, Gavriliu and Boncalo [15] focuses on reducing memory consumption, which is achieved by recalculating previous values, instead of storing them for reuse. The throughput of this design is again 1 pixel/cycle, with a clock speed comparable to [12] (225 MHz). For the same design parameters, this implementation will have identical latency to any of the other three designs, but logic use will be significantly higher (on the order of 4 times as large), and block RAM usage will be around half of that of the others. Table 2.5 shows details of resource usage for the described implementations.

The design parameters shown in figure 2.11 represent a typical configuration of the module. The input window is 3×3, the Harris matrix window is 7×7 and the non-maximum suppression window is 3×3, although all of these may be changed at design time.

The following chapter will discuss what happens with the features once they are detected

Implementation	Memory bits	Register bits
[13]	433920	1729
[12]	276480	1120
[15]	92160	705
This work	341760	1153

Table 2.5: Memory and register usage for some Harris–Stephens implementations. Image size is 1920×1080. For this system, input window 3×3, Harris matrix window 7×7, non-maximum suppression window 7×7, 8-bit input, 16-bit Harris matrix terms, and 33-bit Harris–Stephens scores.

– the extraction of descriptors, which allow corners to be compared against one another, and similarities between them found.

Chapter 3

Feature Descriptor Extraction

3.1 Introduction

As a camera moves through an environment, typically most of the objects visible in one frame will also be visible in the following frame. If the camera has moved, the apparent positions and orientations of the objects may have changed slightly, but in general, the majority of the objects will still be visible.

In order to determine how the apparent positions of objects have changed, it is first necessary to identify which objects in one frame are actually the same objects viewed in the next frame. This can be done by identifying features on the objects that resemble features in the other frame. It is vanishingly unlikely that the regions of pixels around a feature in two successive frames will be identical, and thus a more nuanced approach must be taken to identifying features in two frames that represent the same physical feature. Ideally, we want to calculate for each feature a value that will allow us to match approximately against other values calculated from other features.

3.1.1 Feature Descriptors and Extractors

A ***feature descriptor*** is a value (typically a vector) calculated for a feature using a ***feature extractor***. Feature extractors will typically calculate the descriptor for a feature using the local environment around the feature, since little information is contained in a single pixel. Feature descriptors are structured in such a way that they can be easily compared against other descriptors from other features, in order to find ***feature matches***.

All practically-used descriptors represent a vector in a multidimensional space. Some, such as SIFT (Scale-Invariant Feature Transform, see section 3.2.1), represent a vector of real values in a multidimensional space, and similarity between two descriptors is judged using Euclidean distance. Others, such as BRIEF (Binary Robust Independent Elementary Features, see section 3.2.2) and ORB (Oriented FAST and Rotated BRIEF, see section 3.2.3) are binary descriptors—each dimension of the descriptor is either 0 or 1, and distance between descriptors is judged using Hamming distance, the number of bits that differ in the representations.

In informal speech, it is common to refer to a feature extraction algorithm as a "feature descriptor", although strictly it is the output of a feature extractor that is the feature descriptor.

3.1.2 Invariance

Many feature extraction algorithms are described as being ***scale invariant***, ***rotation invariant***, ***illumination invariant*** or even ***affine transformation invariant***. In practical terms, these mean that the descriptors can be used to match features that differ in the respective way.

Scale invariance means that two features may be matched that differ in scale. This property is particularly useful in SLAM applications, since it allows features to be matched regardless of image depth. A robot might see a particular feature from 10 m away and store its descriptor. When the robot is 5 m away, it sees the same feature again, but this time it has twice the apparent size, thanks to foreshortening. A naive feature extractor may not be able to match these two features, since the scales are so different, but one with scale invariance should be able to.

Rotation invariance allows descriptors to be used to match features that are rotated relative to one another. This particular invariance is less useful for ground-based robotic vehicles, but is extremely useful for airborne robots, particularly ones that observe the ground or bank in order to turn. When observing the ground, the "orientation" of a feature is entirely dependent on the compass orientation of the vehicle, and thus multiple passes over the same ground feature might appear different. Use of a non-rotation invariant feature extractor would cause two descriptors to be significantly different. The same applies to banking—banked flight causes the features to be rotated relative to the vehicle.

Illumination invariance allows an extractor to generate similar descriptors for two views of the same feature under different lighting conditions. Commonly this is achieved by performing calculations relative to a pixel in the feature, and assuming that lighting conditions across

the feature are similar, taking advantage of spacial coherence. A feature extractor without illumination invariance is typically unable to match two views of the same feature under different lighting conditions, for example at two different times of day, or when a light is turned on.

Affine transformation invariance is a property that allows descriptors to be used to match features that have undergone a more complex transformation than simple rotation or scaling, such as a skew or trapezium transformation. Such transformations are less commonly seen in physical objects, but can result from lens distortion around the outer edges of an image.

3.2 Algorithms

This section details a number of the algorithms used for corner-based feature descriptor extraction. Particular attention is given to ORB, which is used in this work.

3.2.1 SIFT

Scale Invariant Feature Transform (SIFT) was introduced in Lowe in 1999. SIFT produces, for a given feature, a 128-dimensional real-valued unit vector representing a histogram of the gradients around the feature point.

The gradient histograms are determined by taking subsets of the pixels around the feature point and determining their image gradients as a vector. The vectors are grouped into 8 bins at 45° spacing, and the sums of the magnitudes of the vectors used to determine the value of the histogram bins.

SIFT provides rotation and scale invariance, and partial illumination and affine transformation invariance. In order to provide rotation invariance, SIFT determines a feature's orientation, and performs all later operations relative to this orientation. This feature orientation is determined by a voting mechanism, where the image gradients at each point in a window around the point contribute to the vote in proportion to the strength of the gradient. The rotation angle is quantised to values of $\frac{\pi}{18}$, or 10°.

Scale invariance in SIFT is achieved by means of an image pyramid (see section 4.1.1). This means that the feature detection and extraction steps are performed at multiple scales, and the results combined together.

The illumination invariance in SIFT is provided by the use of intensity gradients. Since the gradients are calculated as the difference in brightness between two points, assuming that they are subject to the same illumination, the difference in their intensities should be independent of the illumination of the feature.

Limited affine transformation invariance is achieved by selecting the subsets of pixels for gradient histograms such that they represent different distortions of the image. If the distortion is modest, there is a good chance that the gradient histogram will be similar for two different transformations of the same feature.

The SIFT descriptors generated by the extractor are normalised to unit length, and can be

compared to others using Euclidean distance.

SIFT has very good matching accuracy, at the cost of very slow computation and large descriptor size. If stored with 32-bit floating point numbers (the default in OpenCV), each descriptor takes 512 bytes.

3.2.2 BRIEF

Binary Robust Independent Elementary Features (BRIEF) is a feature descriptor extraction algorithm introduced in [18] to address the performance and storage issues of SIFT. The extractor computes, for a feature point, a binary vector representing the results of a number of pixel comparisons in the neighbourhood of the feature.

Each bit of the descriptor is generated by comparing a pair of pixels at predetermined offsets from the feature centre. For example, a pair might be defined as $((-3, 4), (2, -1))$, and the feature might be located at $(100, 100)$. The pixels at $(97, 104)$ and $(102, 99)$ are sampled from the image, and then compared. If the first has a higher intensity than the second, then a 1 is written to the descriptor, otherwise a 0 is written.

The number of pixel comparisons used in BRIEF is variable, but typically between 128 and 512 are performed. The precise set of descriptors is a parameter of the implementation, but is generally unimportant. The authors show good performance with 128 pairs of points chosen randomly with Gaussian-distributed radius from the central point.

BRIEF does not have scale, rotation or affine transformation invariance, just illumination invariance. The illumination invariance comes from using relative comparisons, instead of using absolute intensities. The extractor is not rotation invariant, since a rotation of the input image will cause a different set of pixels to be sampled. Scale invariance can, however, be achieved by using an image pyramid (see section 4.1.1), in the same way as with SIFT and ORB.

BRIEF descriptors are compared using Hamming distance, that is, counting the bits that differ between two descriptors. The matching accuracy is reasonable for cases with minimal rotation, but descriptor calculation and matching performance is much faster than SIFT.

3.2.3 ORB

Oriented FAST and Rotated BRIEF (ORB) was introduced in [4] to address the lack of rotation invariance in BRIEF. ORB uses the same principle of a vector of binary tests, but instead of the

samples being taken at fixed offsets from the central feature point, the samples are rotated to match a canonical orientation of the feature.

The canonical rotation is defined by the angle of a vector pointing towards a point called the ***image centroid***. The image centroid is in turn defined in terms of image moments.

An ***image moment*** is a value defined on a region of an image in a manner similar to a moment of a probability distribution:

$$M_{pq} = \sum_{x,y \in w} x^p y^q I[x, y] \tag{3.1}$$

where M_{pq} is the moment of the image of order p in the x-axis and order q in the y-axis, and w is the window over which the moment is to be calculated, typically a circle or square.

M_{00} is equal to the sum of the pixels in the window and the image centroid is defined as:

$$Centroid = \left(\frac{M_{10}}{M_{00}}, \frac{M_{01}}{M_{00}} \right) \tag{3.2}$$

The centroid is approximately the average location of maximum intensity, so in an image that is mostly black except for a patch of white, the centroid will be located in the middle of the white patch. On real images it will typically be in or near the brightest point, as shown in figure 3.1.

Once the image centroid has been located, the canonical orientation is determined using the atan2(y, x) function

$$Orientation = \text{atan2} \left(\frac{M_{01}}{M_{00}}, \frac{M_{10}}{M_{00}} \right) \tag{3.3}$$

In the case that the centroid is located at the origin, the orientation is undefined. This work discards such cases as degenerate.

In the original ORB paper[4], this angle was quantised to 12° increments (i.e. 30 different angles) in order to enable the use of precalculated lookup tables for the samples. The version of ORB currently in OpenCV now has an accuracy of "about 0.3 degrees"[19], corresponding to approximately 1024 angles, and no longer uses multiple lookup tables.

The lookup table for the sample coordinates used in ORB has them stored relative to an orientation of 0°. Before the samples are drawn from the image, their coordinates are rotated by

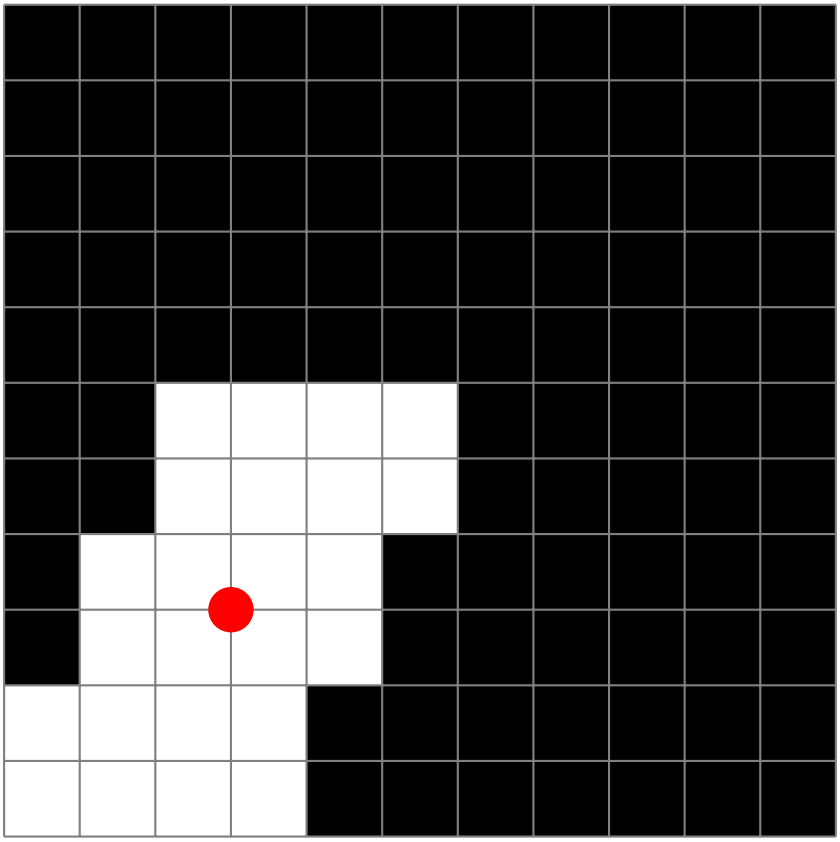


Figure 3.1: The image centroid for a feature

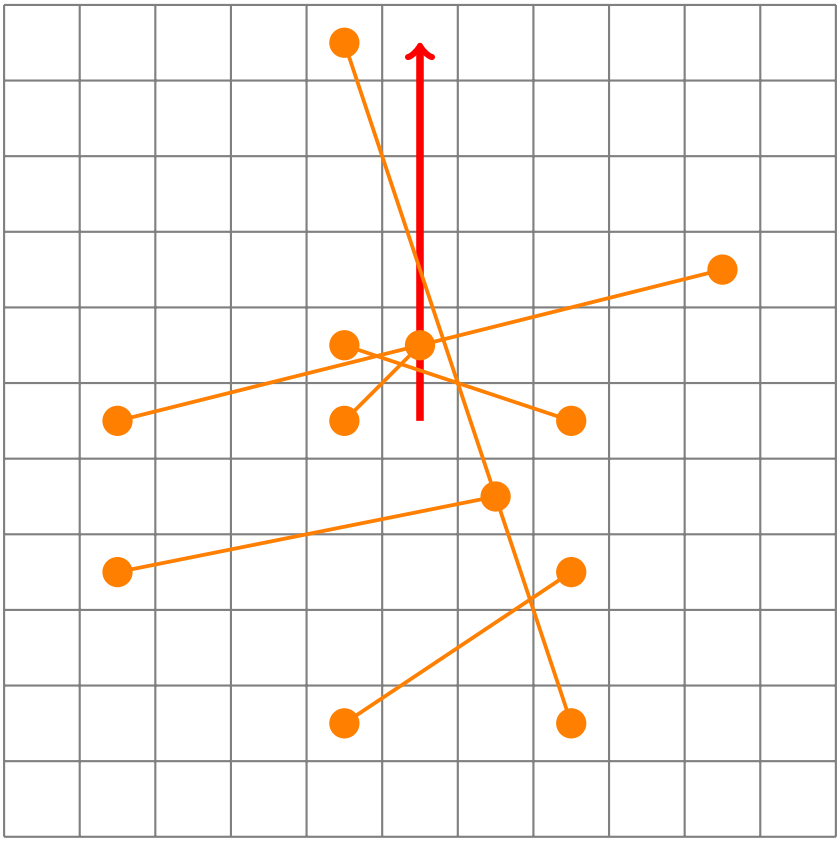


Figure 3.2: Some ORB samples

the negative of the feature's orientation, to align them with the rotated descriptor. This is shown for a few pairs of samples in figure 3.3.

Since the samples are rotated according to the orientation of the feature, ORB is significantly more invariant to rotations of the input image, and will successfully match in such cases. Additionally, in an improvement upon BRIEF, the sample points are chosen with more care than simple random selection.

According to the paper[4], a large number of pairs of samples was generated, and then these were ranked in order of how much information they provided about the feature. The highest-scoring pair was selected as the first bit, and then the remaining pairs were again ranked by how much additional information they provided, before selecting the pair for the second bit. This process was repeated until 256 pairs had been selected.

Since only high-information pairs are selected, this has the effect of reducing the correlation between different bits in the binary descriptor, and generally improves the information content of the 256 bits that ORB returns for a feature.

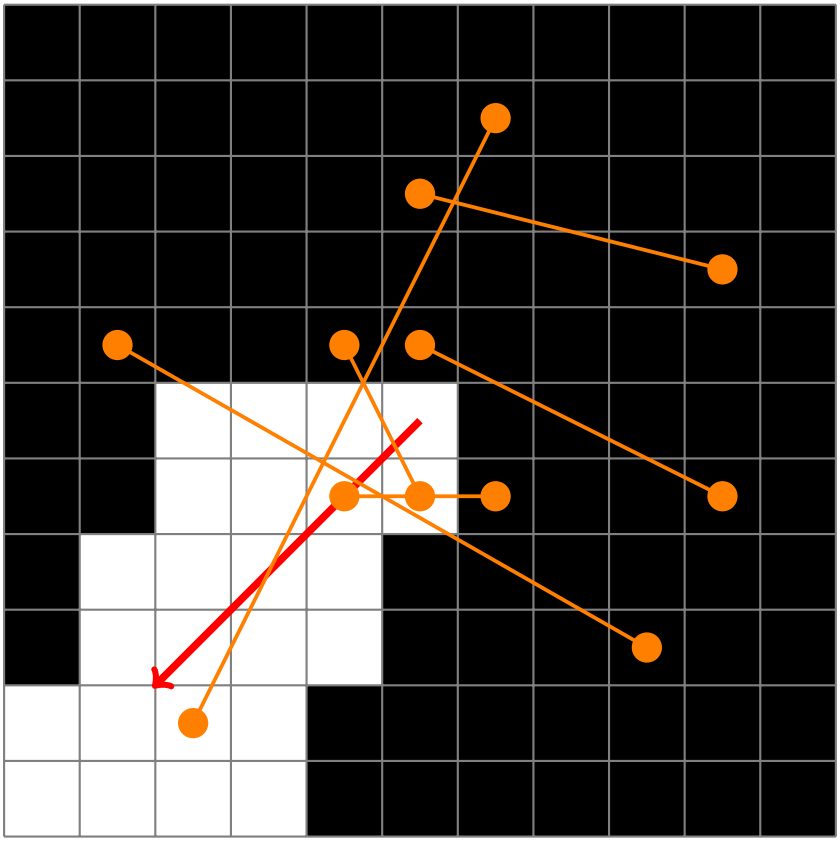


Figure 3.3: The ORB samples from figure 3.2 rotated to align with a feature

ORB has rotation invariance, as described above, and illumination invariance, in the same way as BRIEF. Scale invariance can be introduced with the use of an image pyramid, as discussed in section 4.1.1. General affine transformation invariance is, however, minimal, since distortion of the image will change the pixels sampled.

ORB has generally good performance, a little slower than BRIEF because of the extra image moment calculation and rotation steps, but fast enough for realtime video processing of small images. The matching performance is, however, significantly better than BRIEF, particularly for image pairs where the transformation involves a rotation. ORB is significantly faster than SIFT, with comparable matching performance[4].

Since there were few high-performance implementations of ORB at the outset, and since it shows good matching performance, ORB was selected as the feature extraction algorithm to use in this work.

3.3 Previous Work

3.3.1 SIFT

Bonato, Marques and Constantinides [20] developed an FPGA-based implementation of SIFT in 2008, using a Difference-of-Gaussians feature detector. The algorithm is otherwise a close translation of the CPU implemention onto the FPGA platform. Performance was reported as 30 fps for 320×240 pixel images, a throughput of 2.3 MPix/s.

Yao, Feng, Zhu, Jiang, Zhao and Feng [21] developed a higher-performing implementation in 2009 by making small modifications to the algorithm to better conform to the FPGA environment, including reducing the length of the vector of floats output as the descriptor. Performance was reported as 32 fps for 640×480 pixel images, a throughput of 10 MPix/s.

3.3.2 BRIEF

The BRIEF algorithm is much better-suited to FPGA implementation than either SIFT or ORB, since it does not involve any expensive rotation operations, meaning that a sliding window can be used for both feature detection and feature extraction, and direct access to the pixels in the feature extraction window can be performed. This means that BRIEF descriptors can be calculated using a pipeline with unity throughput, and that the only barrier to performance is clock speed.

de Lima, Martinez-Carranza, Morales-Reyes and Cumplido [22] developed a hardware implementation of BRIEF in 2015, using specially-selected BRIEF sample points to maximise performance. The work does not detect corners, but instead assumes that they are already detected and in a memory, ready to use.

The BRIEF sample points are arranged in such a way that all the necessary pixels can be sampled using 22 4-byte reads. The memory the authors use is capable of unaligned reads, and the BRIEF sample tests are selected such that they only use pixels from these 22 read operations. This allows the pixels to be read using random access relatively quickly. Once loaded into registers, the pixel comparison tests can be executed rapidly using fixed addresses.

Since the design does not have to detect corners, the throughput is best measured in terms of features processed. This architecture manages one 256-bit descriptor every 15 clock cycles.

This, in theory, means that for a unity-throughput feature detector, it should be able to extract one BRIEF feature descriptor every 15 pixels. F_{max} is reported as 125 MHz.

Hu and Ikenaga [23] developed an extremely fast implementation of BRIEF in 2017, which processes four pixels per clock cycle on average to achieve a throughput of 400 MPix/s at 100 MHz, equivalent to 1300 fps at 640×480, on a Kintex-7 FPGA. The BRIEF points are not randomly selected, but chosen in such a way as to maximise the information content, in a similar way to ORB.

3.3.3 ORB

Lee [6] developed an FPGA-based ORB implementation in 2014. The design uses FAST as the corner detector and appears to store the image in an external RAM.

The method used for image storage in the FAST detector is unclear from the paper, but it appears that it is loaded from some external storage into a 7×7 internal memory, which might be a sliding window, or possibly just a wide RAM.

FAST corner detection proceeds in a standard manner, doing the pixel comparisons. It appears that the corner detection has a maximum throughput less than 1 pixel/clock, possibly due to the time taken to fill the memory. If a point is determined to be a feature, its address is pushed into a FIFO queue for feature extraction.

A second submodule is used to calculate the ORB descriptors for points. It appears that pixels are fetched from the external memory once again and fed into a 31×31 block-RAM (BRAM) memory. The image moments are calculated from this BRAM, processing one column of data at a time. The orientation is computed using an atan2 lookup table.

The descriptor bits themselves are calculated using four copies of the window in parallel. The coordinates are transformed according the orientation of the feature, and samples taken from the window. The parallel modules allow multiple bits to be calculated in parallel, shortening the total calculation time.

The paper is unfortunately short on detail, and skims over many of the complex aspects of the design, such as the time taken to load the windows, the external memory access patterns, and the design of the pixel samplers.

The design was implemented on an Artix-7 FPGA, running at 100 MHz. Throughput is reported at 33 MPix/s for 640×480 images with 600-800 features, or approximately 100 fps. The

design is able to process around 80000 features/s.

3.4 Implementation

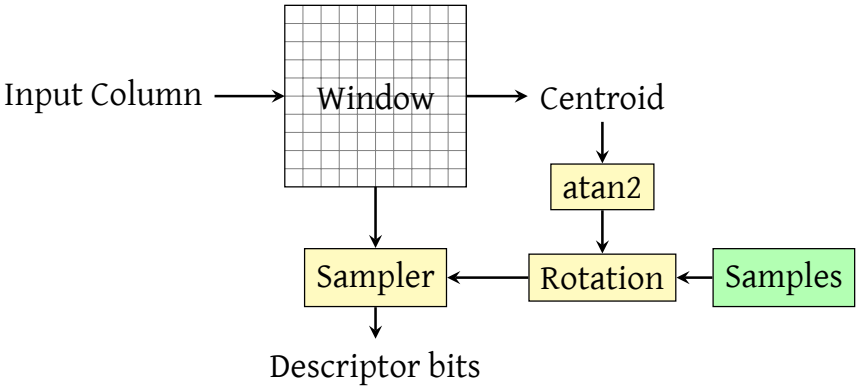


Figure 3.4: The ORB Module [2] ©IEEE 2017

This section details the FPGA implementation of ORB feature extraction. The input to the component is a stream of single pixels and an accompanying feature signal from a feature detector, and the output is a stream of coordinate/descriptor pairs.

This system has been designed with the goal of achieving maximum throughput. It is capable of handling input of one pixel every clock cycle, and thus is able to keep up with the feature detector. However, since the extraction of ORB feature descriptors takes longer than 1 clock cycle, the extractor is not necessarily able to keep up with all of the detected corners. The approach taken in this work is to drop corners that cannot be processed in real time.

Owing to the use of non-maximum suppression in the feature detector, the maximal density for corner detection is one every two pixels (two adjacent pixels cannot both have the highest score in their 3×3 windows). The extraction of a descriptor can take up to 266 clock cycles, and thus for an input of 1 pixel/clock cycle, it is possible for 132 features in every 133 detected to be dropped. A feature density of $\frac{1}{2}$ is thankfully rare in real-world images, and so typical drop rates are significantly lower.

In addition, it is possible to have multiple independent instances of the ORB module running in parallel, so if one module is busy processing a feature and another is detected, a second module can handle the second feature, avoiding a drop. The addition of additional modules will of course provide diminishing returns, as regions of low density will lead to many modules sitting idle, and each module has a substantial logic and memory cost. A more detailed analysis of this trade-off is presented in section 5.4.

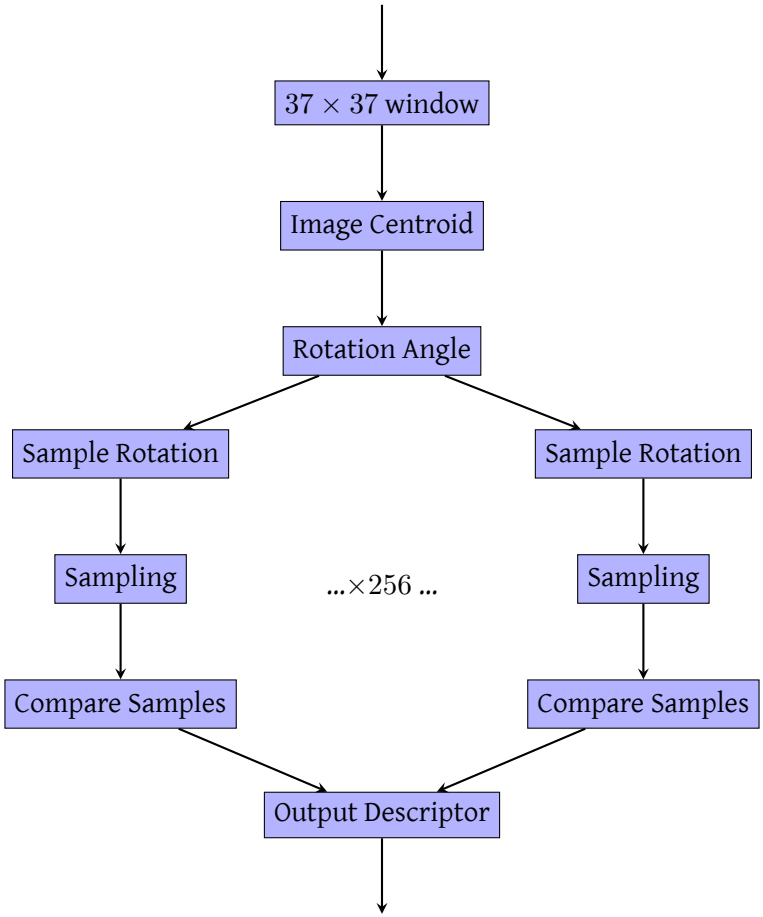


Figure 3.5: ORB data flow and dependencies

The ORB algorithm has two primary processing steps. The first is the calculation of the image centroid, in order to determine the rotation of the sample mask. The second is to sample pairs of points from the feature environment and compare them to generate descriptor bits.

As shown in figure 3.5, the calculation of the rotation angle is a precondition on performing any of the samples. The determination of the image centroid is a precondition on calculating this angle, and this requires all the pixels of the 37×37 window. Once the angle is determined, however, all the samples could potentially occur in parallel.

3.4.1 Feature Orientation

The rotation angle is calculated as

$$\theta = \text{atan2}\left(\bar{y}, \bar{x}\right) = \text{atan2}\left(\frac{M_{01}}{M_{00}}, \frac{M_{10}}{M_{00}}\right) \tag{3.4}$$

Since hardware divisions are so expensive, we would like to remove the division by the patch sum M_{00}. Conveniently, atan2 is independent of scale, and thus

$$\theta = \text{atan2}\left(\frac{M_{01}}{M_{00}}, \frac{M_{10}}{M_{00}}\right) = \text{atan2}\left(M_{01}, M_{10}\right) \tag{3.5}$$

This result means that we only need to calculate M_{01} and M_{10}, then use them as inputs to some realisation of atan2 (see subsection 3.4.1).

Image Moments

In order to calculate the rotation angle, we need the the first image moment in the x- and y-axes (M_{10} and M_{01} respectively) for a region around the detected feature. Image moments are defined as:

$$M_{pq} = \sum_{(x,y)\in w} x^p y^q I[x, y] \tag{3.6}$$

where $p, q \in \mathbb{Z}$ and w is a windowing region around $(0, 0)$.

This work uses w as the region $\{-18, -17, \ldots, 17, 18\}^2$, representing a 37×37 window centred about 0, since this is the rectangular bound from which the ORB samples are drawn. In theory, any size region could be used to determine the moments — a Gaussian window centred at the origin, or a circular window with antialiased edges, would be two options worthy of consideration. This work uses a rectangular region since it greatly simplifies the calculation, and doesn't significantly affect the detected corner orientation.

Proposal 1: Gigantic adder tree

The naive approach to calculating these two desired moments would be to construct two extremely large adder trees, one for each moment, with 1369 input elements, one for every element

of the window. This tree would need to be 11 levels tall, and each input would require a constant multiplier representing either its column or row address, as appropriate. Such a structure would impose a latency of at least 12 cycles, and consume a very large fraction of the available routing logic.

Proposal 2: Integral images

An alternative approach is to use a structure like an ***integral image***, introduced in [24], to accelerate the calculation of the image moments. An integral image is an array of values the same size as the input image, where the value at each coordinate represents the sum of the values in the rectangle defined by that coordinate and the origin, in the input image.

$$ii[x, y] = \sum_{x' \leq x, y' \leq y} i[x', y'] \tag{3.7}$$

where i is the input image and ii the integral image.

Integral images can be used to calculate the sum of the pixels in any arbitrary axis-aligned rectangle of the image in constant time, independent of the size of the rectangle, using the relation

$$RectSum((a, b), (c, d)) = ii[c, d] - ii[a, d] - ii[c, b] + ii[a, b] \tag{3.8}$$

where (a, b) and (c, d) specify two opposite corners of the rectangle.

It is possible to generate an integral image on the fly, using a sliding window. Since the integral image values are stored in the cells of the sliding window, the size of the rectangular sum is limited by the size of the window. An integral image generated over a 37×37 sliding window would have sufficient information to enable calculation of M_{00}.

It is possible to calculate the sums of each row and column of the image moment window using this 37×37 integral image. The sums are then multiplied by their respective offset from the centre of the window to enable calculation of M_{01} and M_{10}.

The sliding window necessary to implement this integral window must be very high-

precision. For a 1920×1080 image with 8 bits per pixel, the bottom-right array cell needs 29 bits of storage, and thus a 37-row-high sliding window would require over 2 Mb of block RAM. By taking advantage of the commonalities between successive image moments, this approach can be improved upon, significantly reducing the memory requirements, as described in the next section.

Proposal 3: Iterative computation

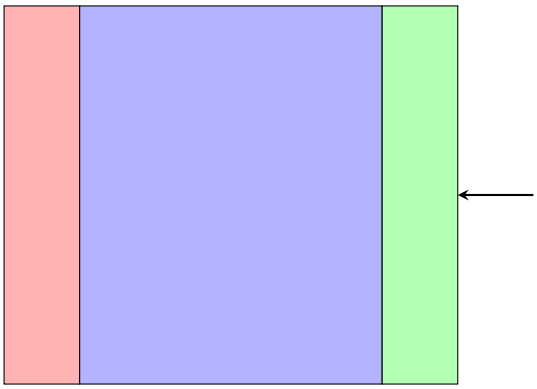


Figure 3.6: At each step, a column of input pixels is added to the right side of the window, and a column of old pixels is removed from the left side, leaving the window size constant.

The approach employed in this work makes use of iterative computation for both the x- and y-moments. The ORB window can be considered as a sliding window moving across the image. Each update step, one column of pixels from the image is added to the right-hand side of the window, and one column of pixels from the left-hand side of the window is removed. Iterative calculation of the image moments then becomes a function of the incoming column of pixels and some internal state, instead of requiring access to the value of every pixel in the window.

X-Moment

(Lindsay Kleeman contributed to the analysis in this section) Consider the x-moment case. We assume that we have some internal state for a given 37×37 window, and we want to find the moment for the next window. Formally, we want to find a function for

$$M_{10}|_{k+1}\left(S_k, input_k\right) \tag{3.9}$$

where S_k is the current state and $input_k$ is the incoming column of pixels.

In our particular case, with a 37×37 window, we have

$$M_{10}|_{(k_x,k_y)} = \sum_{x=-18}^{18} \sum_{y=-18}^{18} x \cdot I[k_x + x, k_y + y] \tag{3.10}$$

where k_x and k_y are the x- and y-coordinate of a particular point, k. Likewise,

$$M_{10}|_{(k_x+1,k_y)} = \sum_{x=-18}^{18} \sum_{y=-18}^{18} x \cdot I[k_x + x + 1, k_y + y] \tag{3.11}$$

For the sake of generality, we define h as the half-window size; the full window has dimension $(2h + 1) \times (2h + 1)$. For the 37×37 window, $h = 18$.

In addition we define a new function,

$$\text{colsum}(x, y) = \sum_{a=-h}^{h} I[x, y + a] \tag{3.12}$$

Using these definitions, M_{10} can be expressed as:

$$M_{10}|_{(k_x,k_y)} = \sum_{x=-h}^{h} x \cdot \text{colsum}(k_x + x, k_y) \tag{3.13}$$

Subtracting $M_{10}|_{(k_x,k_y)}$ from $M_{10}|_{(k_x+1,k_y)}$ gives:

$$M_{10}|_{(k_x+1,k_y)} - M_{10}|_{(k_x,k_y)} = \sum_{x=-h}^{h} x \cdot \text{colsum}(k_x + 1 + x, k_y) - \sum_{x=-h}^{h} x \cdot \text{colsum}(k_x + x, k_y) \tag{3.14}$$

$$= \sum_{x=-h+1}^{h+1} (x - 1) \cdot \text{colsum}(k_x + x, k_y) - \sum_{x=-h}^{h} x \cdot \text{colsum}(k_x + x, k_y) \tag{3.15}$$

Expanding the first term

$$M_{10}|_{(k_x+1,k_y)} - M_{10}|_{(k_x,k_y)} = \sum_{x=-h+1}^{h+1} x \cdot \text{colsum}(k_x + x, k_y) - \sum_{x=-h}^{h} x \cdot \text{colsum}(k_x + x, k_y) - \sum_{x=-h+1}^{h+1} \text{colsum}(k_x + x, k_y) \tag{3.16}$$

The first two terms in equation 3.16 have significant overlap, and thus they cancel out partially

$$M_{10}|_{(k_x+1,k_y)} - M_{10}|_{(k_x,k_y)} = x \cdot \text{colsum}(k_x + x, k_y)\big|_{x=h+1} - x \cdot \text{colsum}(k_x + x, k_y)\big|_{x=-h} - \sum_{x=-h+1}^{h+1} \text{colsum}(k_x + x, k_y) \tag{3.17}$$

Substituting we get

$$M_{10}|_{(k_x+1,k_y)} - M_{10}|_{(k_x,k_y)} = (h+1) \cdot \text{colsum}(k_x + h + 1, k_y) + h \cdot \text{colsum}(k_x - h, k_y) - \sum_{x=-h+1}^{h+1} \text{colsum}(k_x + x, k_y) \tag{3.18}$$

The third term in equation 3.18 is equivalent to $M_0|_{(k_x+1,k_y)}$, that is to say, the sum of the elements of the next patch. We can expand it to transform the term into $M_0|_{(k_x,k_y)}$, i.e. the current patch sum:

$$\sum_{x=-h+1}^{h+1} \text{colsum}(k_x + x, k_y) =$$
$$\sum_{x=-h}^{h} \text{colsum}(k_x + x, k_y) - \text{colsum}(k_x + h + 1, k_y) + \text{colsum}(k_x - h, k_y) \quad (3.19)$$

Substituting back into equation 3.18, we have

$$M_{10}|_{(k_x+1,k_y)} - M_{10}|_{(k_x,k_y)} =$$
$$h \cdot \text{colsum}(k_x + h + 1, k_y) + (h + 1) \cdot \text{colsum}(k_x - h, k_y) - \sum_{x=-h}^{h} \text{colsum}(k_x + x, k_y) \quad (3.20)$$

This expression allows us to calculate the change in x-moment cheaply, only requiring the sum of the column of pixels entering the window, the sum of the column of pixels exiting the window, and the sum of all the pixels in the window:

$$\Delta M_{10}(k+1) = 18 \cdot \sum(\text{incoming pixels}) - 19 \cdot \sum(\text{outgoing pixels}) - M_{00} \quad (3.21)$$
$$= 18 \cdot \sum_y \text{input}_k[y] - 19 \cdot \sum_y \text{input}_{k-37}[y] - M_{00}|_k \quad (3.22)$$

Reformulating the above into the discrete state-space representation of equation 3.9, we have:

$$\boldsymbol{S}_k = \left(M_{10}|_k, M_{00}|_k, \sum \boldsymbol{input}_{k-1} \ldots \sum \boldsymbol{input}_{k-37}\right) \quad (3.23)$$
$$M_{00}|_{k+1} = M_{00}|_k + \sum \boldsymbol{input}_k - \sum \boldsymbol{input}_{k-37} \quad (3.24)$$
$$M_{10}|_{k+1} = M_{10}|_k + 18 \cdot \sum \boldsymbol{input}_k - 19 \cdot \sum \boldsymbol{input}_{k-37} - M_{00}|_k \quad (3.25)$$
$$\boldsymbol{S}_{k+1} = \left(M_{10}|_{k+1}, M_{00}|_{k+1}, \sum \boldsymbol{input}_k \ldots \sum \boldsymbol{input}_{k-36}\right) \quad (3.26)$$

Thus, the state required to be maintained between steps is:

- the current x-moment, M_{10}

- the current sum of all pixels in the window, M_{00}
- a shift register of the sums of the previous 37 columns of input

The new output x-moment is a function of the previous x-moment, the current input, the input from 37 steps ago, and the sum of the pixels in the window.

Y-Moment

The calculation of the y-moment is more straightforward. Again, an iterative computation is used, but because the window moves in the x-axes, a given pixel always has exactly the same weighting in the y-moment for the duration of its time in the window.

The sum of the pixels in each row of the window is tracked as the window moves across the input image, and these sums are multiplied by their y-coordinate before being summed to yield the y-moment value (M_{01}).

$$M_{01}|_{(k_x,k_y)} = \sum_{y=-18}^{18} \sum_{x=-18}^{18} y \cdot I[k_x + x, k_y + y] \tag{3.27}$$

In state space

$$\boldsymbol{S}_k = \left(rowsum_{-18} \dots rowsum_{18}, \boldsymbol{input}_{k-1} \dots \boldsymbol{input}_{k-37}\right) \tag{3.28}$$

$$rowsum_n|_{k+1} = rowsum_n|_k + \boldsymbol{input}_k[n] - \boldsymbol{input}_{k-37}[n] \tag{3.29}$$

$$\boldsymbol{S}_{k+1} = \left(rowsum_{-18} \dots rowsum_{18}, \boldsymbol{input}_k \dots \boldsymbol{input}_{k-36}\right) \tag{3.30}$$

$$M_{01}|_{k+1} = \sum_{y=-18}^{18} y \cdot rowsum_y \tag{3.31}$$

The state required to be maintained between steps is:

- the sums of all the pixels in each of the rows of the window
- the previous 37 columns of input

Unlike the x moment calculation, the full delayed input column is required, not just the sum of the pixel values. Thus instead of a shift register, the x moment calculation reads a column

directly out of the ORB window (see section 3.4.2).

Arctangent

The original ORB paper[4] suggests quantising corner rotation angles to one of 30 values, representing increments of 12°. However, the code currently (at the publication of this work) available on OpenCV GitHub has switched to using a higher-precision representation, with an error of "about 0.3 degrees"[19]. The precision of the angle is in fact a free parameter, and can be chosen to optimise some criterion.

The approximate angle calculation is done by quantising the angle to the centroid into one of 2^n ***sectors***, where n is the number of bits. Sector 0 points directly upwards (in the negative y direction in image coordinate space) and sector numbers increase in the clockwise direction. The angle for a sector k is given by:

$$\theta = \frac{2\pi k}{2^n}, k \in \{0, 1, \ldots, 2^n - 1\} \tag{3.32}$$

Angle 0 is defined to point upwards (negative y direction) and angle $\frac{\pi}{2}$ to point right (positive x direction).

Figure 3.7 shows a number of runs of the Tarsier simulator using different precisions of the sector calculation. As can be seen, for coarse angle divisions, matching performance is low, since the algorithm is effectively functioning as BRIEF. When the number of rotation angles is very high, the results tend to stabilise, and no extra performance is attained by adding finer divisions. In the middle, around 2^6 sectors, the number of matches found is significantly higher. This is attributed to false matches, where two corners that should not be matched are similar enough to match.

This work chooses to quantise to 64 sectors, as this number offers a good compromise between accuracy and reduced complexity of implementation, relative to 1024 angles.

The arctangent module itself is designed to calculate a 6-bit (64-value) approximation of the angle towards the centroid. The first two bits represent the quadrant of the 2-dimensional space that the centroid lies in, which is easily calculated by examining the signs of the x and y arguments to the module. Once the quadrant has been identified, the centroid coordinates are rotated so that they lie in the top right quadrant of the plane, as shown below:

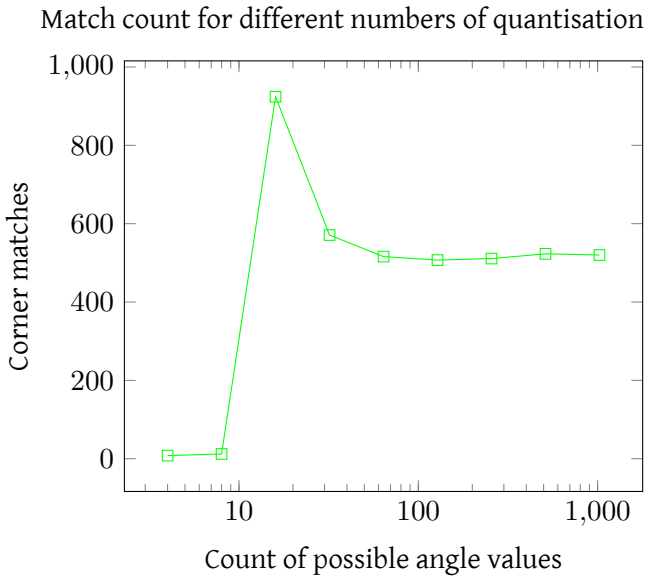


Figure 3.7: A plot of the number of corner matches for different degrees of angle quantisation, matching the same pair of images

$x > 0$	$y > 0$	Quadrant	New x	New y
no	no	top left	$-y$	x
no	yes	bottom left	$-x$	$-y$
yes	no	top right	x	y
yes	yes	bottom right	y	$-x$

If both x and y are 0, the orientation is undefined, since the centroid lies at the origin. In this case, the feature is degenerate, and cannot be processed, thus is dropped.

The remaining four bits depend on the angle within the quadrant. The following analysis examines the top right quadrant and using only four angles per quadrant, for the sake of simplicity. Figure 3.8 shows this quadrant and the five sectors that cover parts of it. The results can be generalised to the full unit circle.

The problem is thus reduced to determining which sector a given point lies in. One possible approach is to map the centroid point onto the line $x = 1$, using similar triangles. The mapped y value could then be compared against the values of the blue sector boundaries at $x = 1$, and the appropriate sector determined by finding the highest boundary less than the y value.

$$y_{mapped} = \frac{y}{x} \tag{3.33}$$

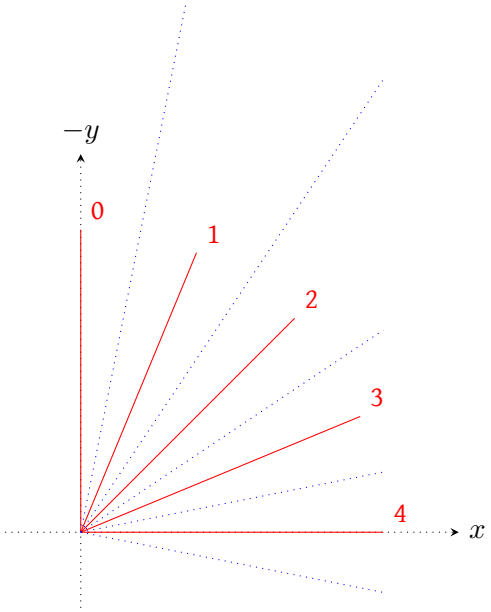


Figure 3.8: The sectors of one quadrant. Red lines mark possible output angles. Blue dotted lines mark the boundaries between different sectors. If a centroid lies within a sector, the red line represents the angle that will be returned by the atan2 module. This diagram represents a configuration with four angles per quadrant, thus 4 bits of angle precision across the full unit circle.
Note that the coordinates are in image coordinate space, thus positive y is downwards.

Sector	Upper bound at $x = 1$	Upper bound at $x = 174$
0	-5.027	-874.8
1	-1.497	-260.4
2	-0.668	-116.3
3	-0.199	-34.6
4	0.199	34.6

Table 3.1: The comparison table for a centroid at $x = 174$

The downside of this approach is that mapping the centroid onto the line requires a division operation, which is slow and logic-intensive. However, the approach can be salvaged by instead calculating the comparison table at $x = 1$ and then using similar triangles to shift it to the appropriate x coordinate.

Consider a feature with a centroid at $(174, -212)$. Since the x coordinate is 174, the intercept comparison table must be scaled by a factor of 174, as shown in table 3.1. The y coordinate is compared against each of the scaled upper bounds, and the feature belongs to the first sector with an upper bound greater than the y coordinate. In this case, the centroid is in sector 2.

This algorithm can be implemented without requiring any division in 5 clock cycles, as shown in *SectorSel.sv*.

3.4.2 ORB Window

Structure

The ***ORB window*** is a memory structure specifically designed for the access patterns needed in ORB. ORB requires random read access to pixels in a 37×37 window, and the window needs to be able to be kept up to date as new image data arrives. The window is able to load image data in the same way as a sliding window, with a new column of pixels entering the window every clock cycle, and an old column exiting to make space. Additionally, the window provides the data necessary for calculating the X and Y moments of its contained pixels iteratively.

The window is realised as a very wide ***ring buffer*** in block RAM. A ring buffer is a memory where input data is written into consecutive addresses. When the write pointer of the ring buffer reaches the end of the memory, it begins again from the beginning. This means that the ring buffer stores a history of the elements written into it. Earlier values are stored at lower addresses.

The ORB window is implemented as a ring buffer of width $37 \times 8 = 296$, and length of 64, the smallest power of two greater than 37. On each update step, the window is loaded with a column of 37 pixels, which replaces the oldest word in the memory. This column then becomes the rightmost column of the window contents. The 37 addresses preceding the current value of the write pointer represent the contents of the window, with columns to the right having higher addresses than those to the left. As each new column is written, the left column of the window leaves the window, but remains in an unused portion of the memory until it is later written over by new data.

The column of new input pixels comes from a source external to the window. In the system as implemented, this takes the form of a sliding window that also provides the pixels for the Harris–Stephens detector.

The structure of the window means that the effective location of the window rolls around the ring buffer, and so the memory address of a given window coordinate must be calculated dynamically by adding an offset given by the address of the write pointer. It is possible for the window to be split into two parts—one at the end of the memory, and one at the beginning, with unused memory in the middle. Since the memory has length of a power-of-two, it is straightforward to use wrapping arithmetic to calculate a given address. The window coordinates are defined such that the origin of the window is in the middle of the 37×37 square. The most recently input

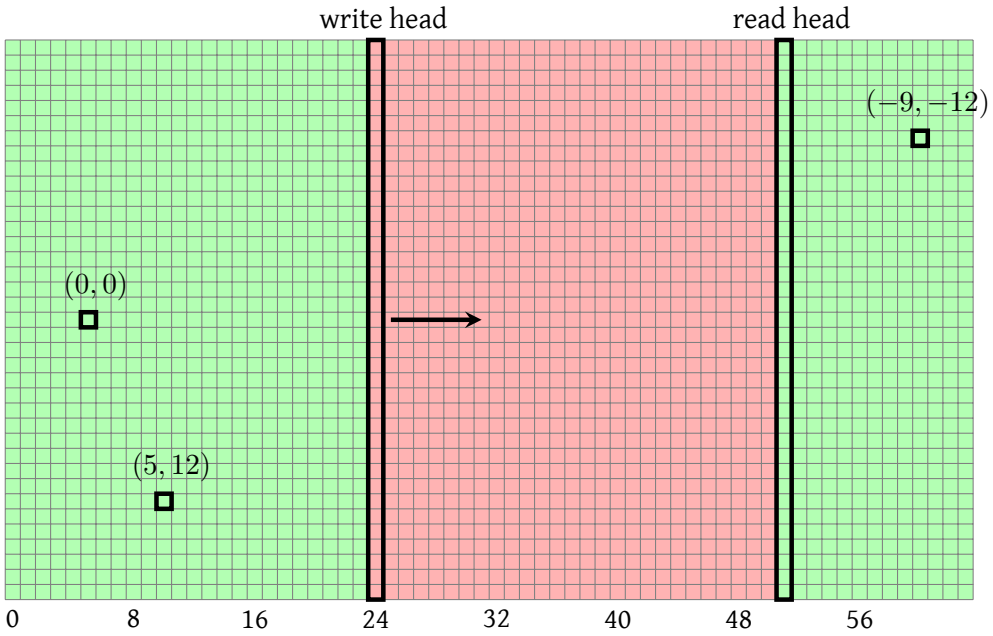


Figure 3.9: A depiction of the state of an ORB window in memory. The origin is at address 5, the write head is at address 24, and the read head is at address 51. The current window contents are shown in green and unused space in red.

column is at $x = 18$, and the oldest column has $x = -18$. y coordinates are defined according to image convention, so the bottom of the window has $y = 18$, and the top has $y = -18$.

As an example, if the write pointer into the ORB window is currently at address 24, the column of pixels at address 23 has $y = 18$. Thus, the column containing the origin (at $x = 0$ is at address 5. The column with $x = -18$ was written before the write head wrapped around, and is located at address 51. The space between addresses 24 and 50, inclusive, contains old pixel data which is no longer needed. This is shown in figure 3.9.

When a new column of pixels is written into the window, it will be written into address 24. The write pointer will then advance to address 25 and the origin to address 6.

The ORB window is implemented in *ORBWindow.sv*.

Access

The ring buffer is implemented in dual-ported block RAM, which constrains the access patterns to the ORB window. Each of the ports can operate as a read port or as a write port, and both are used in different operation modes of the window.

The ORB window operates in two modes—write mode and read mode. In write mode, columns

of input are written into the window and the moments are updated. Whenever a column of input is written, the write head advances one position. Write mode is the mode used to keep the ORB window in sync with the stream of input.

Read mode on the window is used when a corner has been detected, and the ORB module must access pixels for the comparisons that generate the descriptor. In read mode, write updates to the window cease, and thus the window can become out of sync with the input pixel stream. While in read mode, each of the two ports operates as an independent random access port, allowing access to any pixel in the window. This means that two pixel accesses can occur concurrently, constituting one ORB comparison per clock cycle.

It is necessary to freeze the ORB window when a corner is detected, since otherwise the window's contents would soon be overwritten by new input columns. When ORB descriptor generation finishes, and the ORB window returns to write mode, it is possible that input columns have been received without being written to the window. Since this implementation does not have the facility to replay previous input, it is then necessary for the ORB window to wait until a full 37 columns of input have been received without interruption before the window is once again fully valid. Owing to pipeline delays in the moment calculation modules, this takes 46 input columns in the current implementation.

The validity status of the window is managed by a ***dirty flag***. When the ORB window module transitions into read mode, the flag is cleared, marking the window as being clean. If a column of input is received while the module is in read mode, it cannot be included in the window, and must be discarded. This means that if the window later resumes writing, it will have missed some input, and there will be a gap. Thus when the input is received, the flag is set, indicating the window is dirty. If the dirty flag is set when the window returns to write mode, it will take a full refresh of 46 columns of input before the window is once again valid and ready to transition back to read mode.

Moments

The ORB window also provides the raw data for the calculation of image moments. The X moment calculator only requires the column of input pixels, which is passed in without modification. The X moment module will sum the pixels and feed them into its own shift register.

The Y moment calculator, on the other hand, requires a shift register of a window's width

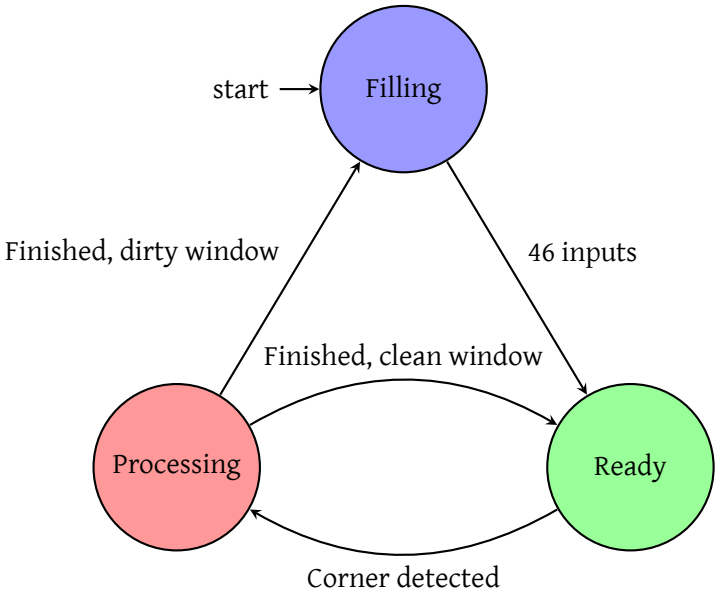


Figure 3.10: State machine for each ORB module

of input columns, so that it can remove values from its row sum registers once they leave the window. The only value actually required from the shift register, however, is the outgoing pixel column, which is always resident in the ORB window. Since only one of the two memory ports is used when the ORB window is in write mode, the other memory port can be used as a read port pointing to the column of pixels that will leave the window on the next update step, thus the ORB window itself serves as the shift register needed for the Y moment module. The read head location is shown in figure 3.9.

3.4.3 ORB Module

The ORB module is responsible for coordinating one or more ORB windows and actually doing the comparisons for the ORB descriptor. An ORB module contains ORB windows, moment calculation modules and an arctangent module.

An ORB module can be in one of three high-level states—***filling***, ***ready***, or ***processing***.

The ***filling*** state represents the condition where the module's ORB windows have not yet received a full window's worth of consecutive image pixels. This may be because the current location is at the beginning of the image, or because input was missed during a recent descriptor generation. In this state, the windows are all in write mode, allowing them to remain in sync with newly-received pixel columns. Since the windows are not ready, the module is unable to

process any features that are detected.

After a full window's worth of pixels have been received, the ORB module transitions into the ***ready*** state. While in the ready state, the ORB module is ready to begin processing a feature to extract its descriptor when triggered. The windows remain in write mode, and the image moments are valid. The arctangent module also provides valid output in this state, so feature orientation is continuously calculated, even if the pixel in question is not determined to be a feature. When triggered by a feature detector, the module transitions into the processing state.

In the ***processing*** state, the module proceeds to calculate the bits for the ORB descriptor. In this mode, the ORB windows are in read mode, permitting two random access pixel reads per clock cycle. The ORB windows can be duplicated to allow multiple comparisons to be executed in parallel. One module allows one comparison per clock, two modules allow two comparisons, and so on. While the ORB module is in the processing state, it is unable to respond to further detected corners. Once the descriptor generation is finished, the module will transition to either the filling state or the ready state, depending on whether the ORB modules are dirty or clean, respectively.

Calculation of the sector to which the centroid belongs imposes a delay of 6 clock cycles. Once this result is available, it is used as an index into a precomputed table of sin and cos values. These trigonometric tables provide the values used to transform the sample coordinates used in the ORB descriptor from their unrotated positions into the orientation of the feature. For an unrotated sample (x, y) rotated by angle θ, the rotated sample coordinates (x', y') are given by:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.34}$$

This calculation is performed using signed fixed-precision arithmetic and the final values are rounded. The trigonometric lookup table values are precomputed to 8 bits of precision, and the rotation operation imposes a latency of one clock cycle. This rotation is implemented in *VectorRotate.sv*.

Once the rotated coordinates are obtained, they are used to address the read operations in the ORB windows. Each read has a latency of one clock cycle, and the read values from the two samples are compared together to provide one bit of the descriptor. These are shifted into a

256-bit serial-to-parallel converter for output.

The total computation time for the ORB module is dependent on the number of ORB windows operating in parallel. In the case of one window per ORB module, the delay is 266 clock cycles. For n modules, the total computation time is given by

$$time = 10 + \frac{256}{n} \tag{3.35}$$

The implementation of the ORB module can be found in *ORB2.sv*.

3.4.4 ORB Arbitrator

Since ORB modules are only able to process one feature at a time, if another feature is detected while a module is still generating the descriptor for the previous one, the module cannot handle it and it must be dropped. This will result in a reduced proportion of detected features having their descriptors extracted successfully, which may lead to impaired matching performance.

To reduce the impact of this dropping, multiple modules can be implemented in parallel. This way, if a feature is detected while one module is still processing, another module can take its place and handle the newly-received feature. An arbitrator is needed to coordinate the behaviour of these multiple ORB modules.

The ORB arbitrator module is responsible for delegating each received feature to one of its subservient ORB modules. It keeps track of the state of each of the modules, and feeds each detected feature to the next available module in the ready state. In addition, it collects the output from each of the ORB modules and returns this to the remainder of the system. Since only one module can be triggered every clock cycle, and descriptor generation time is fixed, it is only possible for one module to return a descriptor during a given clock cycle, so there is no need for internal buffering.

The implementation of the ORB arbitrator can be found in *ORBArbitrator.sv*.

3.4.5 Design Choices

Why not just store the whole image in RAM?

The ORB windows take a lot of resources. Could one not just store the whole image in a big buffer and do random access on it?

The Tarsier device in its default configuration is able to process images up to 4K (3840×2160 pixels). This represents almost 71 Mbit to store a full frame in on-chip block RAM (row lengths must be a power of 2). The largest and most expensive Stratix V FPGAs, at the time of publication, have approximately 52 Mbit of block RAM, and so are not able to store even one frame, let alone the whole image pyramid.

For 1080p images (1920×1080 pixels), merely 18 Mbit of storage is sufficient, which is the entire block RAM complement on the Arria V GX used for implementation. It is effectively impossible to store images of any useful size entirely in block RAM.

Moreover, block RAM only has two ports. Random access on a whole image stored in block RAM would be limited to one module at a time, completely negating the benefits of multiple ORB modules operating in parallel.

Why not just copy pixels into the patch when needed?

As an alternative to storing the whole image in block RAM, it could be stored on off-chip DRAM and feature neighbourhoods streamed in when needed. This architecture is very similar to the CPU or GPU approach, where the image itself is stored in main memory, and random accesses to the image are stored in a local cache.

The disadvantage of streaming from external RAM is the latency. DDR3 latency is approximately 50 ns[25], and 667 MHz [26] DDR3 is able to stream at 1333 MT/s, although the transfer rate will be limited by the module clock speed, approximately 150 MHz. 370 32-bit word reads are necessary to transfer a full patch of image pixels, which will take at least 370 cycles. 370 cycles is significantly longer than the time taken to compute a descriptor with the current system (266 cycles).

Why is the window 37×37, and not 27×27?

The published sample points of the ORB feature extractor are confined to a 26×26 window centred approximately at the point $(0, 0)$ and ranging from -13 to 12 in both axes. The samples are spread over the whole of the square (points $(-13, -13)$, $(-13, 12)$, $(12, -13)$ and $(12, 12)$ are all sampled).

When these samples are rotated, however, the corners are now rotated out of the 27×27 bounding window. The point at $(-13, -13)$ is the furthest from the centre, at a radius of

$\sqrt{13^2 + 13^2} \approx 18.38$, and thus when rotated by $\frac{\pi}{4}$, would be located at $(0, -18)$ (after rounding). Because of this, the sampling window must be of radius 18 at least.

It would be possible to implement this using a circular window, but for simplicity, this work uses a square window of 37×37 pixels, covering the range from -18 to 18 in both x and y axes.

Could one not load multiple samples simultaneously?

It could be argued that the implemented ORB window structure wastes considerable memory bandwidth. When in read mode, each row of the window is effectively an independent dual-ported 8-bit RAM, but yet we use the same address for all the rows on each port, and only take one byte from the 37 bytes read on each port. Since each row has its own address bus, theoretically a design could read $37 \times 2 = 74$ pixels from the ORB window every clock cycle, and finish all the memory accesses in only 7 clocks.

The ORB algorithm as described in [4] is conducive to an easy implementation of the above. The limiting factor for this approach is the number of distinct samples drawn from each row of the image. Since the two ports on each row can only feed 2 pixels per clock cycle, any row with more than 14 pixel samples would require longer than 7 clocks.

As it happens, there are only 374 distinct pixel samples in the ORB descriptor, and they are reasonably evenly distributed between row -13 and 12, with a minimum of 7 samples per row and a maximum of 22. Thus, all the accesses required for an unrotated ORB descriptor could be performed in only 11 clock cycles. This is, of course, a huge improvement over the 256 clock cycles required to perform random pixel access in the current design.

However, when rotation is taken into account, the situation becomes significantly more complicated. Because different samples are at different radii from the central point, when their coordinates are rotated, two samples which were previously on the same row can be located on different rows, with the effect that the sequencing of the accesses is dependent on the rotation. With 64 different rotation angles, it becomes highly impractical to orchestrate all the different access patterns.

One possible approach would be to implement a ***reorder buffer*** (ROB), similar to that used in an out-of-order superscalar CPU architecture. The ROB would contain the samples yet to be performed, and processors for each memory row could take memory addresses from the ROB and perform the samples in an out-of-order manner. In effect, we would have 37 distinct processing

units, each capable of dispatching two operations per clock cycle.

The performance of this approach is highly dependent on the size of the ROB. A ROB of size 2 has identical performance to the approach presented in this work, while requiring minimal crossbar logic. A ROB of size 374 (the number of distinct ORB samples) would have optimal speed, because each row would be immediately occupied and never starved of input. The cost of such an approach is the crossbar switch required to route coordinates to their target rows, and samples to their comparators. While not impossible, such a switch would occupy so much of the hardware resources as to be intractable.

3.5 Other Work

Lima, Martinez-Carranza, Morales-Reyes and Cumplido [7] is a modification of the authors' previous work, [22], turning the BRIEF extractor into an ORB extractor. Like its predecessor, this work assumes that the features (and now their orientations) have already been detected, and are stored in a list in memory.

The paper describes using an identical set of pixel samples as in the BRIEF paper. The pixel comparisons only use 88 pixels, which are clustered into 22 4-byte read operations, two of which can occur in parallel. Instead of just translating the sample addresses so that they are in the neighbourhood of the detected feature point, this work rotates the sample points to coincide with the detected feature alignment.

If the random accesses were single-pixel reads, this would be a reasonable and effective strategy. Unfortunately, because the reads are chunks of four pixels, adjacent to one another in the x-axis, this is a problematic approach, since some pixels are rotated around the incorrect centre point (see figure 3.11). This can cause a pixel sample to be offset by up to 7 pixels from its correct location.

Performance is identical to the BRIEF implementation—each descriptor can be extracted in 15 clock cycles. For feature matching with minimal rotation, this implementation will perform well, but for cases with significant rotation, it is expected that this will have severe consequences.

Sun, Liu, Wang, Accetti and Naqvi [8] is a more conventional approach to designing an FPGA ORB extractor. Features are detected using a sliding window-based FAST detector, and are then queued in a FIFO. A separate module handles determination of the image centroid and orientation, and calculation of the descriptors.

An image pyramid (see section 4.1.1) is used to allow multiscale processing, with a $\frac{4}{5}$ scale factor between layers.

When the extractor module reads a feature location from the FIFO, it begins by loading the relevant pixel area from external (DDR3) memory. The window loaded is 31×31 pixels by default, but certain rotations require a larger window, and in these cases, the extra pixels are loaded after the orientation is calculated.

The orientation itself is calculated as the image enters the window, using a radius 15 circle. This is a more accurate approach than the square window used in this work, but is computation-

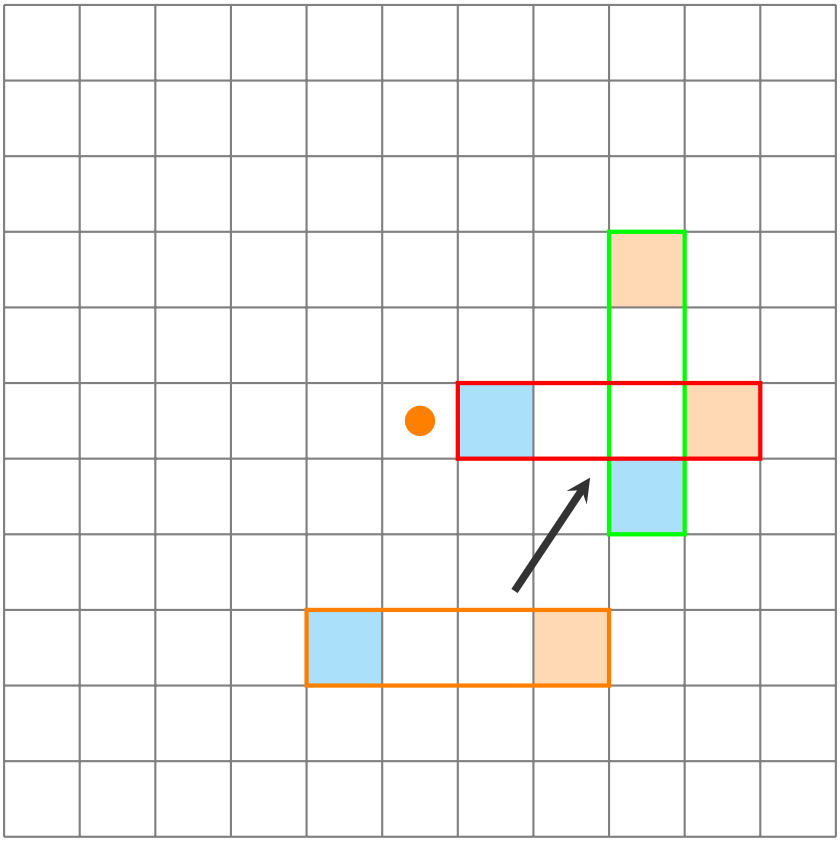


Figure 3.11: The ORB feature extractor in [7] performs its samples in horizontal blocks of four, which leads to significant positional discrepancy when rotated. The orange rectangle shows two samples (blue and orange) at canonical orientation. The green rectangle shows the "correct" position for these two samples, if they are rotated by 90°, according to the ORB algorithm. However, the approximations used in [7] mean that instead, the samples are taken from the locations shown in the red rectangle, a significant distance from their correct locations.

ally more complicated, and does not allow easy iterative calculation. A CORDIC[27] algorithm is used to determine the orientation from the centroid position.

The descriptor calculation itself is standard, performing one pair of pixel samples and outputting one bit of descriptor every clock cycle.

Performance is quoted as extracting 1000 features from 1080p images at 42 fps, which is equivalent to 42000 features/s or 87 MPix/s.

3.6 Feature Descriptor Matching

In order to do anything useful with image features, they must be matched against corresponding points in other frames. A corresponding point is a point that represents the same real-world object but viewed from a different viewpoint.

ORB descriptors are 256-bit binary vectors, and the distance between two ORB descriptors is measured using the Hamming distance. Hamming distance is defined as the count of bits that differ between two values, and a low Hamming distance indicates a high similarity, and vice versa.

The Hamming distance defines a ***metric space*** on the set of ORB descriptors, since it defines a distance between each point. Unfortunately, the structure of the Hamming metric space makes approximate nearest neighbour matching a hard problem. Some success has been reported with location sensitive hashing (LSH)[4], but the false negative match rate is unacceptably high.

Because of this, almost all ORB implementations use brute-force matching. This involves comparing each descriptor against the full set of candidate features one by one, and recording the lowest Hamming score found. This is a computationally expensive task. Matching a set of n features against m candidates to find correspondences requires $n \times m$ comparisons, each of which involves a wide XOR and a bit population count.

This particular operation is very parallelisable, and well-suited to an FPGA architecture. A very basic brute-force ORB descriptor matcher was implemented as part of this work in [1], but is not currently included in the Tarsier system. It was intended to implement a more powerful version of matching on FPGA, but time constraints meant that this had to be abandoned.

3.6.1 Thresholding and Prominence Matching

When scanning a list of candidate descriptors for the best match, it is common to simply take the feature with the lowest Hamming distance as the best match. However, in some cases, there is no corresponding point, owing to clipping, obstruction, or the feature being dropped due to high load. In these cases, the "best match" might actually represent a completely different object, and will introduce a spurious correspondence, causing inaccuracy in later processing. Many systems will impose a maximum ***threshold***, above which a correspondence is not considered valid. This threshold represents the maximum number of bits that can differ between two descriptors for

them still to be allowed to match.

The precise value of the threshold is a parameter of the system that must be tuned, but values on the order of 40 are common.

A further technique that can be applied to reduce the number of false matches is ***prominence matching***. This tracks not only the best match but also the second-best match, and only considers the match to be a correspondence if the difference between the two best matches is above a threshold. Prominence matching helps to ensure that a correspondence is strong by weeding out cases where many similar features have been encountered. This is common in cases such as repeating patterns, where for instance every window on a wall has the same set of features.

Although prominence matching also inhibits correct matches, in most cases it improves the quality of the matches and removes false positives.

3.7 Conclusion

This chapter has examined several feature descriptor extractors in common use, and investigated a number of hardware implementations of them. ORB was selected as the algorithm for this work, owing to its unexplored state in the hardware acceleration community, its applicability to the FPGA platform, and its discrimination performance.

The implementation of ORB on FPGA offers a number of design decisions—some implementations take the approach of translating the algorithm literally from CPU to FPGA, and keep the same access patterns as a CPU. Others attempt to modify the algorithm to better conform to the hardware platform. This implementation has attempted to create a very high speed system for descriptor extraction, sometimes dropping features in order to comply with real-time constraints.

3.8 Comparison with other work

The implementation of ORB feature extraction in this work is generally higher-performing than other implementations, as far as speed is concerned. There are two main metrics useful for judging speed—pixel rate and feature processing rate. The pixel rate is the rate at which the system can accept pixels, detecting corners and passing them for extraction. It is primarily governed by the feature detector speed. The feature processing rate is the rate at which the system can generate descriptors for features. It is primarily governed by the throughput of the feature descriptor extractor.

The two rates are coupled: a system can only process features as fast as it detects them, and (in some systems) the feature processing elements can apply backpressure to the feature detectors, limiting the pixel rate.

Work	Pixel Rate	Feature Rate
[6]	33 MPix/s	80×10^3 features/s
[7]	n/a	8.3×10^6 features/s
[8]	87 MPix/s	42×10^3 features/s
This work	150 MPix/s	170×10^3 features/s

Of the three implementations with published pixel rates, this work is by far the fastest, since it imposes hard real-time constraints on the processing of frames. Because of this, the pixel rate is 1 pixel/clock cycle, and since the clock frequency is 150 MHz, the pixel rate is 150 MPix/s.

The figure selected for the feature rate of this work is a typical figure for one module. The theoretical maximum for a single module is 563000 features/s, for a situation where one module is buffered and remains clean at all times (these is described further in section 5.4). In this situation, extracting a descriptor takes 266 cycles. [7] reports a performance of one descriptor every 15 clock cycles at 125 MHz. This has been included for comparison, but since the algorithm implemented has been significantly modified from standard ORB to better suit their architecture, it is unclear how relevant the comparison is.

3.8.1 Scale invariance

The implementation presented in this work has not addressed the issue of scale invariance so far. Scale invariance is critical to the performance of a robotic visual navigation system, since it allows a robotic vehicle to match objects viewed from different distances. The standard approach

to achieving scale invariance is the image pyramid, comprising multiple versions of the image at different resolutions. The investigation of an FPGA implementation of an image pyramid is described in the following chapter.

Chapter 4

Multiscale Processing

4.1 Introduction

A highly-desirable property for a feature descriptor to possess is ***scale invariance***. Scale invariance means that two views of the same corner at different sizes should generate descriptors that will match one another. These two views can arise due to the camera zooming in or moving towards the object, for example.

Unmodified ORB does not have the property of scale invariance. The sample pattern is rotated according to the rotation of the corner, giving it rotation invariance, but the pattern is a fixed size, and will always look at the same region of image space around a detected corner. This means that if a corner is detected at a given position, and then the camera moves towards the object, a descriptor generated from the view at the closer position will likely have a descriptor that will not match the previous one.

4.1.1 Image Pyramid

One common method of adding scale invariance to a system is to use an ***image pyramid***[4]. Image pyramids were introduced in a restricted form in [28]. The structure contains multiple copies of an image at smaller scales, generated by downsampling the original image, similar to mipmapping[29] in computer graphics. The original image is considered the base of the pyramid, and each level above is a downsampled version of the level beneath it. The corner detection and feature extraction processes are duplicated for each level of the pyramid and the results are collected at the end. In effect, each level of the pyramid is treated independently of all of the others,

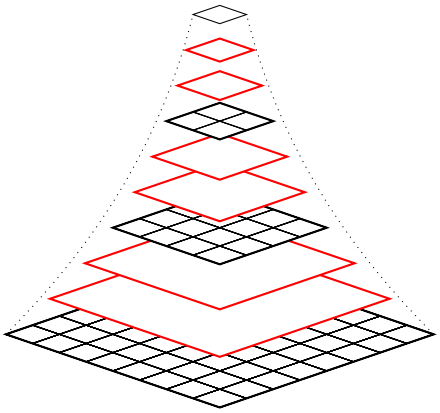


Figure 4.1: Exponential image pyramid. Octave levels are in black, intermediates are in red. This pyramid uses a scaling factor of $2^{-1/3}$ between levels. [2] ©2017 IEEE

as if it were a separate image. If sufficient numbers of downsampled images are generated, it is likely that at least one level will have the feature at a scale close enough to result in a match.

The scale factors are usually chosen to be exponential, so moving a given number of levels up the pyramid results in the same factor reduction of image size. The simplest method is to have each level as $\frac{1}{2}$ the size of the one below in each axis, so it has $\frac{1}{4}$ the number of pixels. Levels are numbered starting with 0 for the base, so if level 0 has k pixels in the x axis, level 1 will have $k/2$, level 2 will have $k/4$, and so forth. Each scaling of $1/2$ is termed an octave, by analogy with music, where an octave represents a doubling of frequency. Typically the number of octaves is a parameter of the system.

Using only one level per octave has acceptable matching performance, as shown in section 5.2, but it can be improved by inserting intermediate levels in-between, maintaining the exponential scaling. With two levels per octave, the scaling factor is $2^{-1/2} \approx 0.707\ldots$ between levels, and with three levels per octave $2^{-1/3} \approx 0.793\ldots$. Adding additional levels increases the likelihood of a downsampled feature matching the target, but comes at the expense of complexity and resource consumption. On a CPU, this takes the form of additional memory consumption and computation; on an FPGA, logic for the duplicated processor chains and block RAM for their sliding windows.

As described further in section 5.2, in general, matching performance increases with increasing numbers of pyramid levels, but with diminishing returns. When matching pairs of images with little scale difference, the impact of the use of an image pyramid is less than it is when matching pairs of images radically differing in scale.

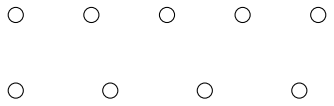


Figure 4.2: The positions of samples in a $\frac{4}{5}$ downsampling operation. The top row shows the positions of the samples in the original resolution signal, while the bottom row shows the reduced density of samples in the lower-resolution output signal.

4.1.2 Resampling

Resampling is the process of representing a discrete signal with different samples. Downsampling reduces the sample rate and upsampling increases it. Downsampling necessarily removes information from the sampled representation, but upsampling does not add it back — information cannot be created.

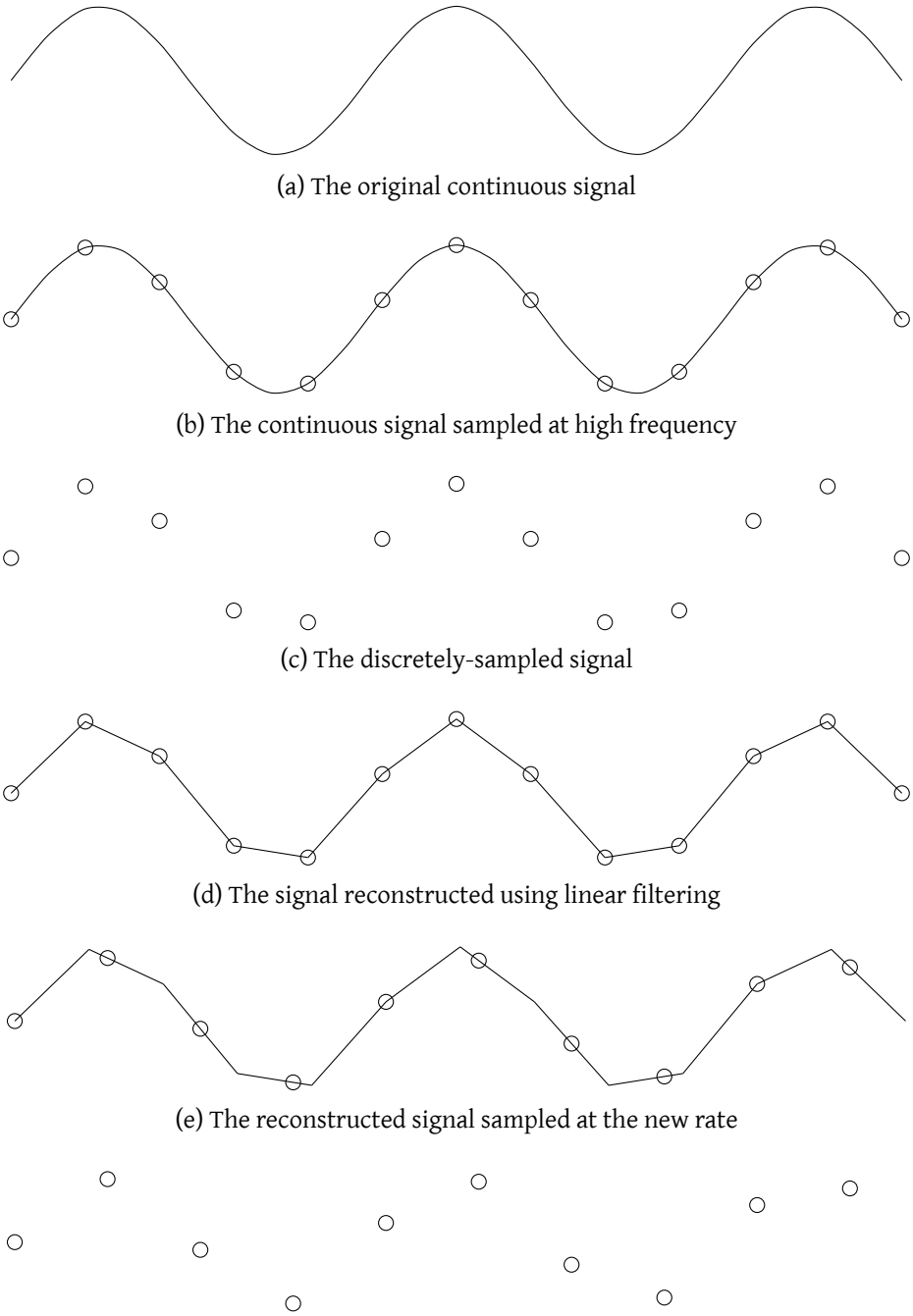
Since each level in the pyramid (aside from the base) is a reduced-resolution version of a level below, a downsampling operation must be performed to yield its image data.

As shown in figure 4.2, most of the output samples in a downsampling operation are coincident with those of the input, and so the signal value is not known at those output sample points. For the $\frac{4}{5}$ downsampling operation shown, no more than one in four output points can be coincident with an input. Determining the value to output at these intermediate points requires reconstructing an approximation to the continuous signal that was originally sampled.

When resampling the input signal, we want to ensure that the resultant samples represent the signal as closely as possible. Nyquist–Shannon sampling [30] ensures that the samples can be used to perfectly reconstruct the continuous input signal, provided that the input signal is ***bandlimited***—that is, its Fourier transform is zero for frequencies above the ***Nyquist frequency***, defined as half the sampling frequency. Any signals above this Nyquist frequency are ***aliased***—after sampling, they are reconstructed at different frequencies below the Nyquist limit.

In order to ensure that the image signal does not cause aliasing when downsampled, it is necessary to apply a low-pass filter to reduce the amplitude of the high-frequency components. This work uses a 2×2 box filter as the low-pass filter for the ratio $\frac{1}{2}$ downsampling operation, which is applied in combination with the resampling by averaging squares of four pixels together. Low-pass filtering has not yet been applied for the ratio $\frac{4}{5}$ downsampling, but this is an area worthy of future investigation.

Figure 4.3 shows how the original continuous image function is sampled by the camera, yield-



(a) The original continuous signal

(b) The continuous signal sampled at high frequency

(c) The discretely-sampled signal

(d) The signal reconstructed using linear filtering

(e) The reconstructed signal sampled at the new rate

(f) The discrete resampled signal

Figure 4.3: Reconstruction of a sampled continuous function for downsampling

ing a discrete signal. The continuous signal is reconstructed using a linear filter, as described in the next section. It is then resampled at $\frac{4}{5}$ the original sample rate, yielding the output discrete signal.

There are a number of different resampling approaches used to reconstruct the original continuous sample for downsampling, but the primary filters used are ***bilinear***, ***bicubic***[31] and ***Lanczos***[32] interpolation, in increasing order of computational complexity. Bilinear interpolation uses the four pixels surrounding a point to provide the image data, interpolating linearly in both x and y axes. Bicubic interplation uses the 16 pixels around the point, fitting a cubic polynomial to the image data in both x and y axes. Lanczos interpolation uses a windowed sinc function as its filter kernel, and in general provides a better-quality resampling, at the expense of computation time. The size of the window is a parameter of the filter – typically 9 to 49 points are used around a point.

4.1.3 Interpolation Filters

The various interpolation methods differ in the process they use to reconstruct the continuous function (see figure 4.4). All of the filtering methods described below are one-dimensional, but they can be extended to n dimensions simply by performing the interpolation in multiple axes at once. By convention, the terms "linear" and "cubic" filtering are reserved for one-dimensional implementations. For higher dimensions, prefixes are added, e.g. "bilinear", "trilinear", etc.

Nearest-neighbour filtering approximates the function by taking the value of the nearest point, effectively dividing the image into Voronoi cells. Halfway between each pair of samples, the interpolated function experiences a step change in value, causing sharp visual artefacts. Computationally, nearest-neighbour interpolation is the simplest, since it only requires one sample and no arithmetic.

Linear filtering fits a straight line between every pair of adjacent points, causing gradient discontinuities at points. Each output requires two samples and a weighted average, so it is slightly more computationally-intensive than nearest-neighbour.

Cubic filtering[31] fits a cubic equation between each pair of adjacent points, with the gradients at the endpoints given by the average gradient in the local neighbourhood, using a curve known as a Catmull-Rom spline. The resultant compound curve is smooth, and generally provides more aesthetically-pleasing results than linear (or bilinear) filtering.

Sinc filtering[30] is performed by multiplying each sample point by a sinc function. The sinc function, defined as

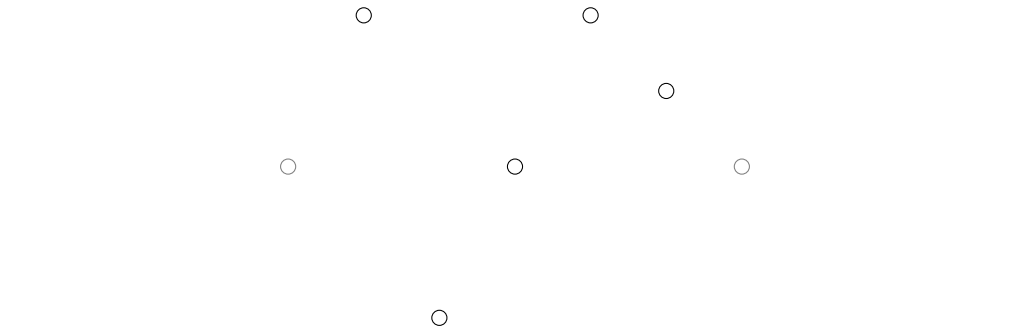
$$\text{sinc}(x) = \frac{\sin \pi x}{x} \tag{4.1}$$

has unity response at $x = 0$ and zeroes at nonzero integer x. When aligned to a discrete input signal, the response will be the sample value at its position, and zero at the positions of all other samples, so the interpolated function uses the sampled value directly. The sampled values are known exactly, and do not need to be interpolated. In between these sample points, the calculated value is a combination of every input sample. The sinc filter is provably [30] the optimal reconstruction filter for bandlimited signals.

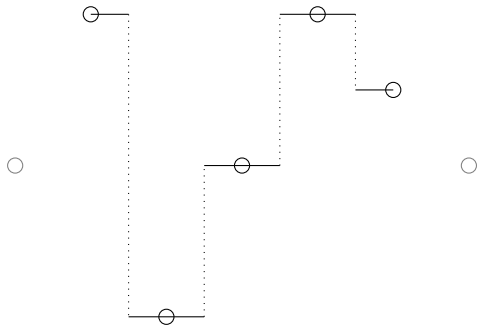
However, since the interpolated value at all non-integer points is a function of *every* sample, the sinc filter is computationally impractical. Formally, the sinc filter has infinite ***support***, where the support of a filter is the number of input values required to calculate an output value. To make the problem more tractable, the ***Lanczos filter***[32] applies a window to the sinc filter, using a parameter a.

$$L(x, a) = \begin{cases} \text{sinc}(x)\text{sinc}(x/a) & -a < x < a \\ 0 & otherwise \end{cases} \tag{4.2}$$

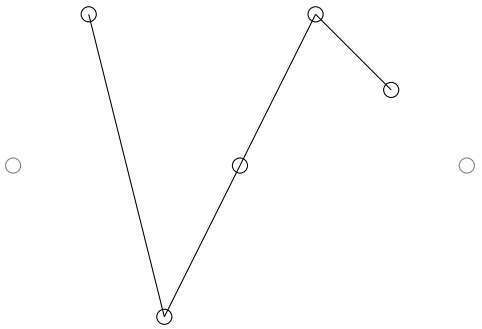
The parameter a is a small integer, usually 2 or 3. The Lanczos filter retains the property of having zero response at all nonzero integer x values, while at the same time restricting the ***support*** of the filter to only $2a$ samples.



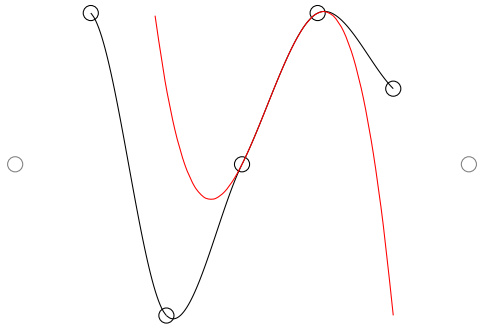
(a) Five input points (plus two adjacent points for interpolation purposes)



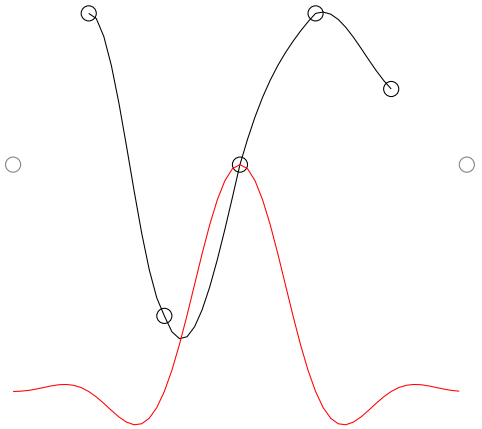
(b) Nearest-neighbour interpolation



(c) Linear interpolation



(d) Cubic interpolation (Catmull-Rom spline)



(e) Lanczos interpolation ($a = 3$)

Figure 4.4: Various interpolation methods applied to the same input points. Some interpolation methods require additional points outside of the interpolated range in order to satisfy smoothness constraints. These are shown as dotted circles.
The red curve in 4.4d shows the cubic curve that is used as the interpolation function between two of the sample points.
The red curve in 4.4e shows one of the scaled Lanczos basis functions that are combined to yield the interpolated curve.

4.2 Implementation

This section discusses the approach taken in integrating a multilevel image pyramid into the Harris–Stephens/ORB system developed in the previous chapters.

4.2.1 Pyramid Structure

The conventional approach for implementing image pyramids in software involves downsampling the images successively in memory. Once all the downsampled versions are available, the feature detection and extraction stages are run sequentially on each of the levels of the pyramid.

This approach is an option for FPGA implementation, but requires a large amount of RAM to store the images—approximately double the image size for an image pyramid with one level per octave, and more than double for more levels per octave. In addition, it fails to take advantage of the parallelism inherent in the process. Each level of the image pyramid is effectively independent of all of the others, and can be processed at the same time without conflict, something for which FPGAs are well-suited.

The input to the feature detection/extraction chain is a stream of pixels in raster order, left to right, top to bottom. An effective multilevel implementation would be able to feed all the levels of the pyramid with their respective pixel streams simultaneously, using a single full-resolution input.

A critical observation is that regardless of which interpolation method is selected, each higher-level pixel is only dependent on a small region of pixels around it in the source image (sinc filtering excepted). This means that as soon as all the pixels from that small region have been received by the FPGA, sufficient information is present to calculate the value of the output pixel. This enables the downsampling to be implemented as a ***finite-impulse response (FIR) filter***, with optional output, greatly simplifying the design.

An FIR filter is a filter whose output is a linear combination of its recent inputs. On FPGA hardware, an FIR is easily formulated as a function applied to the values in a shift register. On each update step, the new input value is fed into the shift register, pushing everything along the chain, and discarding the oldest value. More details on the filter coefficients is in section 4.2.2.

The pixel stream used as input to the Harris–Stephens/ORB feature extraction chain consists

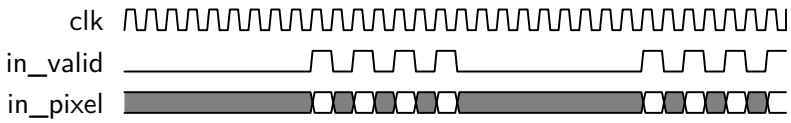


Figure 4.5: Fragmentary pixel stream

of two signals: the pixel value itself, and a valid signal. This valid signal indicates whether the pixel value is valid, and functions as a clock enable for later logic. The valid signal allows a pixel stream to be fragmentary and uneven, as shown in figure 4.5, and supports the optional output from the FIR filters.

Since we are performing downsampling, it is guaranteed that each level in the pyramid has either the same number or fewer pixels than its source level. This further guarantees that the fragmentary pixel stream will be able to keep up with its input, since the input rate is limited to 1 pixel/clock cycle.

There are two main parameters in the design of the structure of an image pyramid: the number of octaves, and the number of levels per octave. The number of octaves is the simpler of the two, since the design scales linearly with the number of octaves. For each octave above the base, its levels are half-scale versions of those in the preceding octave. This work uses 2 octaves, but it is an easily-adjusted compilation parameter.

The number of levels per octave has a significant impact on the design of the filtering, since it determines the sampling ratio between successive levels of the pyramid. As justified in section 5.2, three levels per octave has been selected for this system.

The (linear) resampling ratio between levels of the pyramid is thus given by:

$$2^{\frac{1}{3}} \approx 0.793\ldots \approx \frac{4}{5} \tag{4.3}$$

To simplify the design of the FIR filters, a small integer ratio is preferable, since it reduces the number of different filter coefficients needed. This ratio specifies the size of the repeating tile units that cover the input and output images, and each input tile maps uniquely to an output tile, independent of any other pixels around. A ratio of $\frac{4}{5}$ means that the input tiles are 5×5 pixels, and the output tiles are 4×4 pixels. A ratio of $\frac{1}{2}$ means that the input tiles are 2×2 pixels, and the output tiles are 1×1 pixel.

Each offset in the output tile has a different set of filter coefficients, but the same offset in

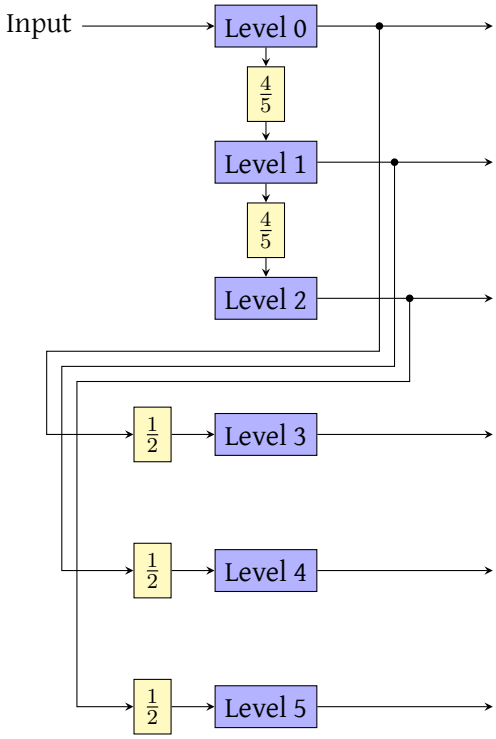


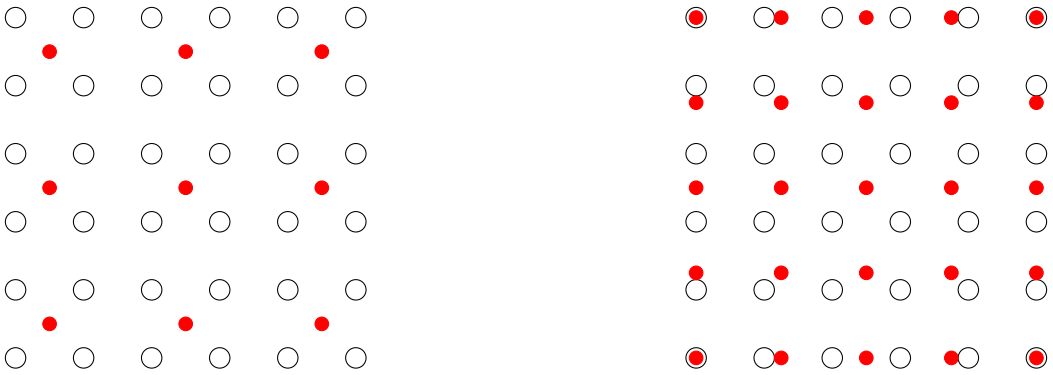
Figure 4.6: The arrangement of scalers used in a three level per octave, two octave image pyramid

each tile has the same filter coefficients.

The ratio used between levels in octave 0 (the base octave) is $\frac{4}{5}$, and between levels in higher octaves and their source levels $\frac{1}{2}$. This cascade of scalers is shown in figure 4.6.

As described below in section 4.2.2, bilinear filtering has been selected for this system. Bilinear filtering requires only the four pixels immediately surrounding a point to have sufficient information to calculate its value. The two filter patterns shown in figure 4.7 have been selected for the two filters used.

In the case of the $\frac{1}{2}$ filter, each input pixel contributes to exactly 1 output pixel, with four inputs contributing equally to each output. Until all four input pixels are received, no output can be generated. Consequently, no output is produced for rows of input with even y value, and no output is produced for input pixels with even x value. The pattern of output that results from this can be seen in figure 4.8, specifically the signals for Level 3. Level 3 is derived from Level 0 passed through a $\frac{1}{2}$ filter. Since output is only produced if the input x and y coordinates are both odd, the output rate is approximately $\frac{1}{4}$, as desired in a half-scaling filter. Note that the output rate is not exactly $\frac{1}{4}$ if one of the dimensions of the input image is an odd number, since some of the blocks of four pixels are incomplete.



(a) $\frac{1}{2}$ filter (b) $\frac{4}{5}$ filter

Figure 4.7: Sample point arrangements for the two downsampling filters. Input: black circles, Output: red dots. [2] ©2017 IEEE

The case of the $\frac{4}{5}$ filter is more complex. Again, each output pixel depends on (up to) four input pixels, but their relative weighting depends on the coordinates of the output pixel modulo 4. The red circles in figure 4.7 represent the positions of the output pixels relative to the input pixels (white circles). As can be seen, each output sample is in a different position relative to its input samples from its fellows.

The red dot in the upper left corner, representing an output coordinate with $x, y \equiv 0 \pmod 4$, is coincident with the input point $x, y \equiv 0 \pmod 5$. This means that it requires only one sample to provide all its pixel information. By extension, all the output points with either $x \equiv 0 \pmod 4$ or $y \equiv 0 \pmod 4$ require only information from two input pixels, and thus the relevant output pixel can be output as soon as those two inputs are available. Input pixels with one coordinate c, where $c \equiv 1 \pmod 5$, however, do not complete the 2×2 input window necessary for an output pixel, and thus no output is produced in these cases. In effect, for the $\frac{4}{5}$ filter, the output is only valid for inputs with $x, y \not\equiv 1 \pmod 5$. This ensures that the output rate approximates $\frac{4}{5}^2 = \frac{16}{25}$, since there are 16 output pixels for every square of 25 input pixels.

This pattern of output is demonstrated in figure 4.8, specifically in levels 1 and 2.

4.2.2 Resampling

Three different scaling methods—bilinear, bicubic, and Lanzcos—were tested in software to determine the best choice for hardware implementation. Two views of the same scene were used as image inputs to two image pyramids, and then correspondences were found using Harris–Stephens feature detection and ORB feature extraction. The method used to interpolate points

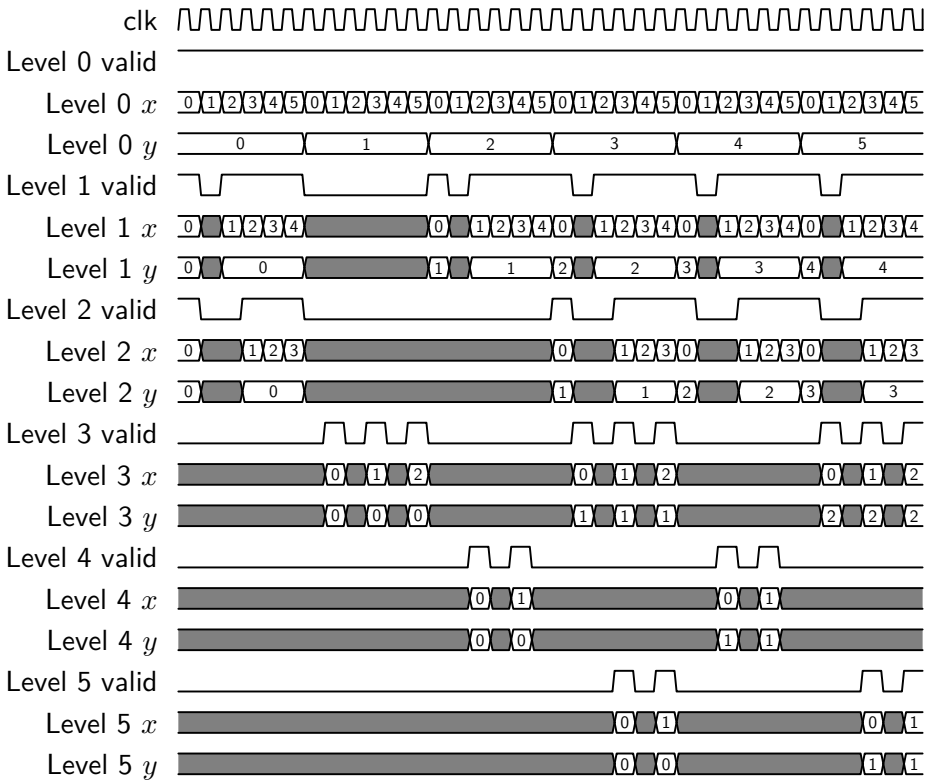


Figure 4.8: Pixel timing for a 6-level pyramid with three levels per octave. Image is 6×6.

Level	Resolution
0	6×6
1	5×5
2	4×4
3	3×3
4	2×2
5	2×2

Table 4.1: The resolutions of the levels in a pyramid with base 6×6. Three levels per octave, two octaves.

for scaling the levels of the image pyramid was varied, and the number of correspondences used as a proxy for matching performance.

No difference in matching performance was observed between the three algorithms used, likely because any difference in pixel values was lost in the quantisation noise.

Bilinear filtering requires a 2×2 window of input pixels, bicubic filtering requires a 4×4 window, Lanczos ($a = 2$) requires a 4×4 window, and Lanczos ($a = 3$) requires a 6×6 window. Since the interpolation performance of the methods is essentially identical for the purposes of image pyramid construction, it was decided to use bilinear filtering, since it is the simplest, and requires the least logic.

4.2.3 FIR Filter

The actual interpolation, for both the $\frac{1}{2}$ and the $\frac{4}{5}$ filters, is implemented using an FIR filter. The FIR filter's output is entirely dependent on a 2×2 window of input pixels, although whether it outputs anything is dependent on the coordinates of the most recent input sample, as discussed above. This is a perfect use for the sliding window design presented in section 2.3.1. The window is $2 \times 2 \times 8$ bits, and has a line width equal to the width of its input level.

The filter convolves the input pixel window with the matrix

$$\begin{bmatrix} b_{0,0} & b_{1,0} \\ b_{0,1} & b_{1,1} \end{bmatrix} \tag{4.4}$$

Since bilinear filtering is being used, for the $\frac{1}{2}$ filter, the coefficients of the filter are

$$b_{0,0} = b_{1,0} = b_{0,1} = b_{1,1} = \frac{1}{4} \tag{4.5}$$

so each element in the window has equal weighting.

Because each of the 16 output positions in a $\frac{4}{5}$ filter's 4×4 output tile has a different set of coefficients, these must be determined based on the offset within the tile. The filter coefficients are fortunately all small binary numbers, simplifying the hardware implementation. They are

given by

$$\text{factors}_a = \begin{bmatrix} 0 & 0 & \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{4.6}$$

$$\text{factors}_b = \begin{bmatrix} 1 & 0 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{bmatrix} \tag{4.7}$$

$$b_{0,0}(x', y') = \text{factors}_a[x'] \times \text{factors}_a[y'] \tag{4.8}$$

$$b_{1,0}(x', y') = \text{factors}_b[x'] \times \text{factors}_a[y'] \tag{4.9}$$

$$b_{0,1}(x', y') = \text{factors}_a[x'] \times \text{factors}_b[y'] \tag{4.10}$$

$$b_{1,1}(x', y') = \text{factors}_b[x'] \times \text{factors}_b[y'] \tag{4.11}$$

where $x' = x \bmod 5$ and $y' = y \bmod 5$.

The formula for the coefficients is slightly complicated by the decision to output the first row and column in each output tile before the whole 2×2 filter contents are available. This means that one more row/column is output in images with dimension d, where $d \equiv 1 \pmod 5$.

The source for these filters can be found in *Scale_1_2_Bilinear.sv* and *Scale_4_5_Bilinear.sv*.

4.2.4 Dimensions

For the $\frac{1}{2}$ filter, if an input image has a dimension x, the corresponding dimension in the output image will be

$$\left\lfloor \frac{x}{2} \right\rfloor \tag{4.12}$$

where $\lfloor x \rfloor$ denotes the "floor" of x—the largest integer less than or equal to x.

For the $\frac{4}{5}$ filter, if an input image has a dimension x, the corresponding dimension in the output image will be

$$\left\lfloor \frac{4(x-1)}{5} \right\rfloor + 1 \tag{4.13}$$

This calculation is performed at the beginning of each frame by a divider. The code for this can be found in *DimensionCalculator_4_5.sv*.

Another useful result is the ***critical pixel*** for a given output. This pixel is the pixel that provides the last piece of information necessary to determine the output value. For the $\frac{1}{2}$ fil-

ter, the critical pixel for a point with coordinate x has coordinate

$$2x + 1 \tag{4.14}$$

and for the $\frac{4}{5}$ filter, the critical pixel has coordinate

$$\left\lfloor \frac{5(x+3)}{4} \right\rfloor - 3 \tag{4.15}$$

4.2.5 Coordinate Transformation

When a feature is detected, the detector only has access to the coordinates of that pixel in its level of the pyramid. Since the levels of the pyramid are of differing sizes, the coordinates of a point in one level of the pyramid will be different to those of the scaled version of that point in a different level of the pyramid. In order to be able to use the coordinates for matching features against one another, all the coordinates must be transformed into the same coordinate frame.

The coordinate frame used is that of the base level of the pyramid. Coordinates of detected features must be transformed before they are returned, and this can be done using one multiplication by a constant, an addition and a shift. For a feature with a coordinate x in level a within octave b, the transformed position is given by

$$\left(2^{b+1}5^a x + \left(2^b - 1\right)5^a + 4^a\right) \ll (2^a + 1) \tag{4.16}$$

This expression includes rounding the feature to the nearest pixel, rather than simply truncating, and is different from the expression for finding the critical pixel.

The implementation of coordinate transformation can be found at *CoordinateTransformer.sv*.

4.3 Conclusion

This chapter has presented a justification of the use of image pyramids, and an efficient hardware implementation of a real time streaming multiscale system. Only two rows of the source image pyramid level must be retained, and so memory consumption is limited to just two rows of image data. In addition, new pixel data flows through the image pyramid as soon as it is available, and so higher levels of the pyramid are not required to wait for completion of the processing of lower levels before they can perform their own processing, resulting in significant speed up.

The pipeline delay of the $\frac{1}{2}$ scaler is only one clock cycle, and only four clock cycles for the $\frac{4}{5}$ scaler, and so the time taken to process a full six-level image pyramid is only 9 cycles longer than processing the bottom level alone. The primary cost of the image pyramid, however, is in the additional logic and memory required for the duplication of the feature detection and extraction hardware. In resource-constrained systems, it might be necessary to process each of the levels sequentially through the same logic, instead of in parallel through many sets of logic.

The following chapter presents results and analysis of the performance of the complete system, and provides justification for a number of the design choices.

Chapter 5

Analysis, Validation and Integration

5.1 Introduction

This chapter opens with an examination of the design parameter space of the hardware implementation, and an investigation of the effects of changing the various parameters of the design on the output. Examples of such parameters that are considered include: bit-widths of Harris–Stephens feature detection intermediate values; number of levels per octave of the image pyramid; and the strategies adopted for buffering input at each level of the pyramid. This is followed by a discussion of the processes used to establish the accuracy of the FPGA implementation of Harris–Stephens feature detection and ORB feature descriptor extraction. The chapter closes with a description of the full Tarsier hardware accelerator device and its integration into the ORB-SLAM2 system to create a demonstrator.

The design parameter space of the hardware implementation is divided into three sections: Harris–Stephens (section 5.3), ORB (section 5.4), and image pyramid (section 5.2). Each division has a number of configurable parameters which influence its performance, in many cases dependent on the contents of the images being processed. Due to the sheer number of parameters, it is not straightforward to compare all of them simultaneously, and so the approach taken is to choose the best performing set in each division and use these when optimising the other divisions.

5.1.1 Simulation tools

The analyses described in this chapter make extensive use of two software simulation tools, which emulate in software the exact arithmetic executed on the FPGA device. The two utilities are written in the Rust programming language[1], and allow the effects of varying design parameters to be simulated easily, without requiring recompilation of the entire SystemVerilog design. The full *Tarsier* design, with 6 pyramid levels and 10 ORB modules, takes approximately 90 minutes to compile on a recent quad-core CPU, emphasising the value in being able to simulate performance before testing on the FPGA.

Sweeper

The first utility, termed the ***sweeper*** (short for parameter sweeper), is able to simulate changes to the feature detection and pyramid construction parameters. This allows one to observe what the results on detected feature counts will be, if one changes the detection threshold, or the structure tensor filter window size. The effects of changing the number of pyramid levels, or the number of octaves, and their effect on the feature counts, can also be observed.

The sweeper is also able to generate descriptors for detected features, but it does not simulate dropping of features; instead, every feature is processed correctly. This allows for comparisons of the descriptors generated by the hardware, to verify that they are correct, but it does not provide a model of which features will be dropped. Descriptor verification is performed by comparing the output of the sweeper against the descriptors calculated by the *Tarsier* for the same images.

A separate python script takes the extracted features and draws matches between pairs of images, allowing visual confirmation of the quality of the extracted features.

Feature Drop Simulator

The ***Feature Drop Simulator*** is designed to perform a fast timing simulation of the ORB modules in a level, allowing one to analyse the effects of changing parameters on the feature drop rate. The simulator allows the module numbers, buffer length and buffering strategy (see section 5.4) to be varied, and the performance observed. It takes an image as input, detects all its corners using the same Harris–Stephens detector used in Sweeper, and then does a clock-by-clock simulation

[1] `https://rust-lang.org`

of the state of the system. On every update step, it determines whether there is an input pixel, if so, whether it is a feature or not, and if it is, attempts to allocate it to an ORB module. The states of the modules are modelled, and if no module is available to receive the feature, it must be dropped.

The drop simulator executes the same state machine as the hardware design, and the correspondence between the two has been verified with the use of SystemVerilog testbench code.

The feature drop simulator will display statistics of how many features are processed and how many dropped, for each level of the image pyramid, and it can also list the exact features processed and dropped.

5.2 Pyramid Structure

The image pyramid has three main design parameters—the interpolation method, the number of levels per octave, and the total number of levels. In general, the more levels in the pyramid, the better the performance will be, since more levels allow extraction of features at more scale levels, increasing the chance that a match will be found successfully. For the same number of octaves, increasing the levels per octave will tend to lead to better matching quality, since the finer gradations of scale mean that there is likely to be a closer match found. However, holding the total levels constant while increasing levels per octave will decrease the total scale range, that is, the ratio between the smallest and largest scale levels in the pyramid, which often has a limiting effect on accuracy.

Since the resource consumption of the system scales with the total number of levels, rather than the number of octaves or the levels per octave, the number of levels will usually be limited by the resource budget of the implemented system. The *Tarsier*, as implemented on an Arria V GX, fits approximately 6 levels, although this could be increased at the cost of resources elsewhere.

This section analyses the performance of a multilevel Harris–Stephens/ORB system as the number of levels per octave and the total number of levels are varied. The analysis is performed using the Sweeper tool 5.1.1. Image pyramids are constructed for each of a pair of images, where the two images are different views of the same scene. Features are detected in these image pyramids using the Harris–Stephens detector, and ORB descriptors are calculated for them, using the configuration shown in table 5.1. They are then matched, and the count of matches is used as the metric to judge the performance of the image pyramid parameters.

The images come from two datasets. The two "kitti_08" images are a stereo pair from the KITTI[2] [33] dataset, sequence 8. The images comprising the "graffiti", "boat" and "bikes" pairs are from the Oxford Visual Geometry Group's Affine Covariant Regions Datasets[3] [34]. The images are shown in figure 5.1.

The four image pairs represent a range of different image pair types. The "kitti_08" images contain almost exactly the same content at the same scale, just viewed from slightly to the side.

[2] `http://www.cvlibs.net/datasets/kitti/`
[3] `https://www.robots.ox.ac.uk/~vgg/data/data-aff.html`

(a) kitti_08

(b) graffiti

(c) boat

(d) bikes

Figure 5.1: The dataset image pairs used in this analysis

Harris–Stephens	
Matrix bits	16
Structure tensor window size	7×7
Score bits	33
Non-maximum window size	3×3
ORB	
Quantisation angles	64
Moment window	37×37
Parallel windows	1
Matching	
Threshold	0
Prominence	20

Table 5.1: Pyramid structure analysis configuration

The "graffiti" images are rotated relative to one another, but hold the scale relatively constant. The "boat" images again have a small rotation, but also a dramatic change in scale: there is a scaling ratio of approximately 3:1 between them. Finally, the "bikes" images are taken from a very similar viewpoint, but with the second image significantly blurred by being defocused.

Figure 5.2 shows how the count of matches for each of the four image pairs changes with different resampling techniques. All of the images show similar results. Lanczos, bicubic and bilinear resampling show consistently good matching performance, and nearest neighbour resampling shows poor performance in comparison. Although it is faster than any of the other techniques, its inaccuracy leads to significantly reduced matching performance.

Of the other three, bilinear filtering is by far the simplest to implement, and requires less logic. Bilinear filtering only requires four neighbouring pixels, where Lanczos and bicubic need 16. Since the performance of bilinear is indistinguishable from Lanczos and bicubic, and the resource requirements are so much lower, this work uses bilinear filtering exclusively.

Figure 5.3 shows how the count of matches for each of the four image pairs changes with different image pyramid construction. The top graph shows a pyramid with 3 levels, the middle with 6 levels, and the bottom with 9 levels. Consequently, 3 levels at 1 level/octave represents the same range of scale as 9 levels at 4 levels/octave. The number of levels is held constant in each graph so that the tradeoff can be seen between fineness of subdivision of scales and total scale range.

The "kitti_08" images show a small improvement in match count with increasing levels per octave at all three level counts. This signifies that in this case, the fineness of scale division is

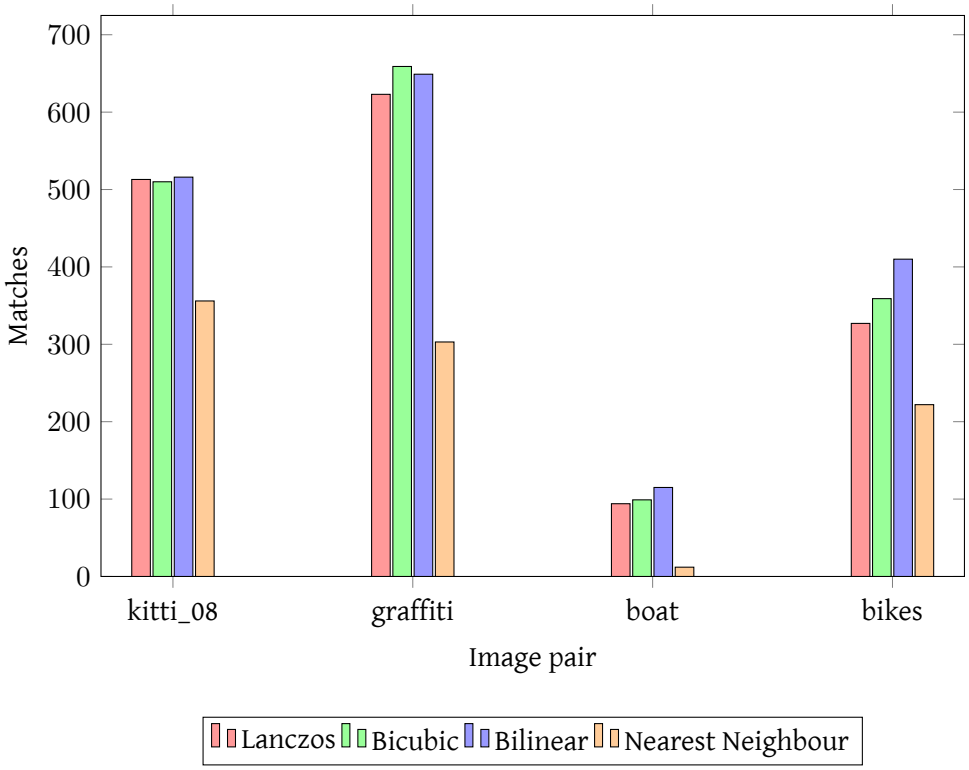


Figure 5.2: Match counts for the four image pairs using different resampling techniques. The pyramid is constructed with 2 octaves each of 3 levels.

more useful than scale range.

The "graffiti" images show a much stronger improvement with increasing levels per octave, but the improvement levels off at 3 levels per octave. This is attributed to the loss of scale range experienced when 4 levels per octave are used, compared to at 3 octaves per level. The reduced step between scales is not as advantageous at this point as having a broader range of scales.

The "boat" images show an interesting effect. With three total levels, no downsampled versions of the zoomed-in image are close enough in scale to the other image to yield any good matches. With one level per octave, this is because the steps between level scales are too big to match with the zoomed-out image. Increasing the levels per octave reduces the scale range further, and means that the image levels are even further apart in scale. It is only once we get to 6 levels at 3 octaves/level that two levels of the pyramids are similar enough in scale to yield some matches. The effect is more pronounced with 9 total levels, and the highest match counts are observed with 9 levels at 4 levels per octave.

The "bikes" images again have difficulty matching when the scale range is very low. The blurred image has few detectable corners at low pyramid levels, since sharp corners have been

blurred out. If the blurred image is downsampled sufficiently, however, effective image sampling rate drops low enough that the low-pass filtering performed by the blur is sharpened, and features can be detected. When both images are downsampled this far, the same features can be detected in both, and matched successfully. As with the "kitti_08" images, increasing the total scale range helps with detecting more features, and so a downturn is observed at 4 levels per octave, where this is reduced.

This work has selected 3 levels per octaves as the ideal number of levels per octave, since it performs better than 1 or 2 levels per octave, while still maintaining a wide total scale range, given limited total levels.

Figure 5.4 shows the results of unrestricted total level count. The same process is performed as in figure 5.3, except that this time, the levels per octave value is held constant at 3, and the total pyramid levels count is increased. All four image pairs show an increase followed by a flattening out, as additional levels stop contributing useful features. The position of the flat region depends on the image pair, however. The "kitti_08" images have reached 90% of their maximum with only 5 levels, while the "bikes" images are still showing improvements at 9 levels.

Given unlimited hardware, an effective design would include as many levels as possible in the pyramid. The six levels that will fit on this FPGA, however, provide adequate performance, although blurry images like "bikes" would benefit most from any additional levels.

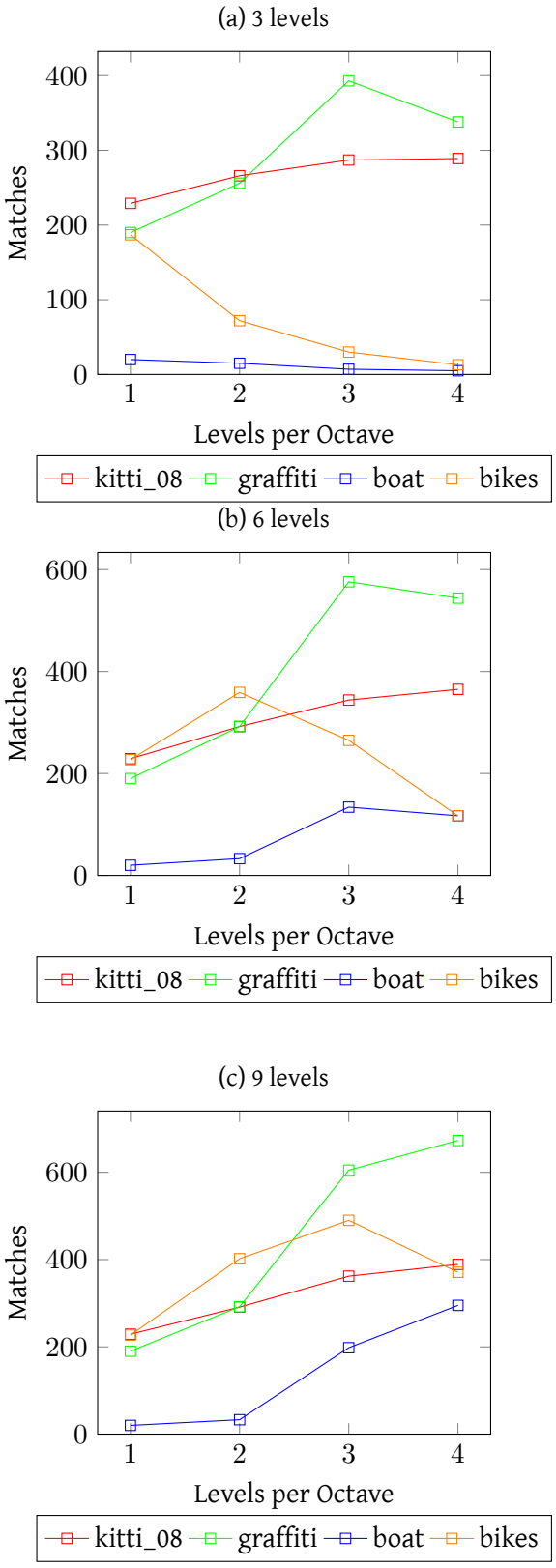


Figure 5.3: Match counts for four dataset images, using varying numbers of total levels and levels per octave.

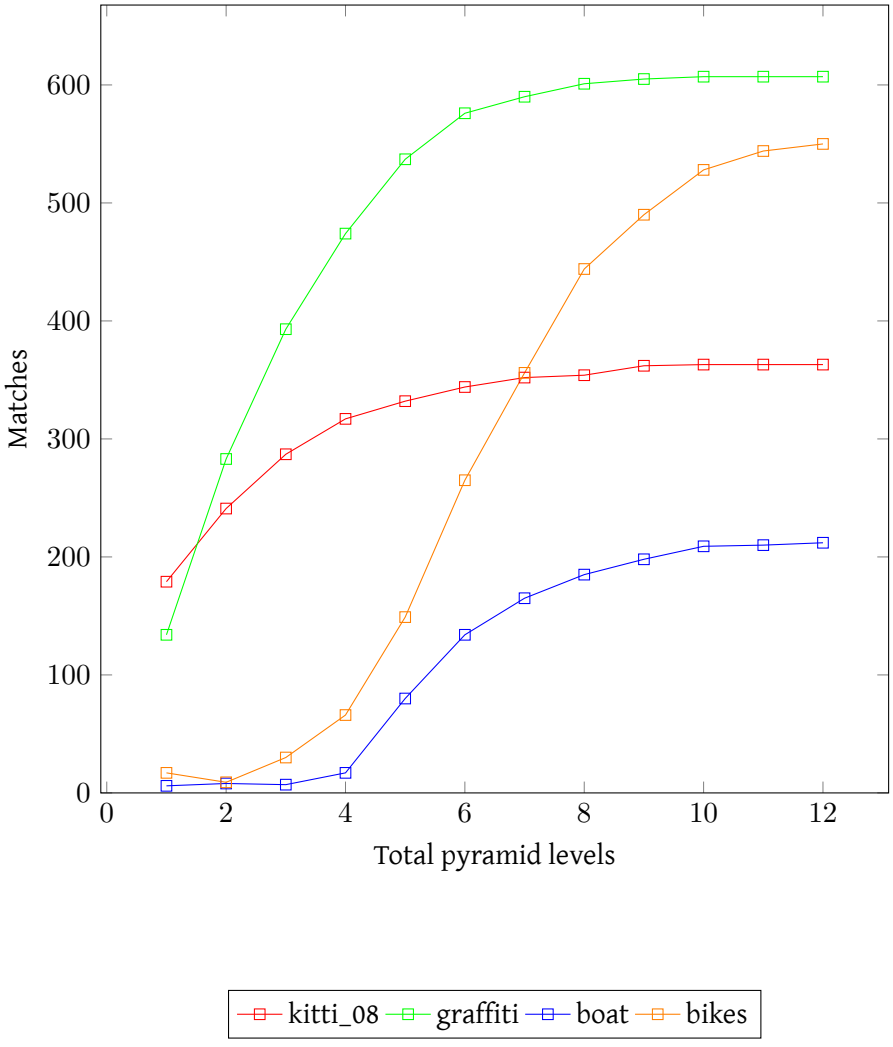


Figure 5.4: Match counts for the dataset images, varying the number of levels while using three levels per octave.

5.3 Harris–Stephens Parameters

This design allows for the configuration of the precision used for intermediate values within the Harris–Stephens feature detector, as well as the precision of the output corner scores. These values are controlled by parameters: `MATRIX_BITS` and `SCORE_BITS`, respectively. The parameter `MATRIX_BITS` specifies the number of bits used to store elements of the Harris matrices and of the structure tensor (see section 2.5.1), while the parameter `SCORE_BITS` specifies the number of bits used to express the output score. Since these values are used in sliding windows, their bitwidth has a significant effect on the memory consumption of the Harris–Stephens detector, and so, if possible, they should be kept as small as possible.

At the same time, the use of the non-maximum suppression filter (see section 2.2.3) means that pixels with exactly the same score as their neighbours may not be considered as corners, thus it is often the low-order bits of the corner score that determine whether a pixel is a feature or not. Particularly low-precision settings of these parameters will thus result in lower pixel counts than higher-precision settings.

This section uses the same four pairs of images as in section 5.2. Using the Sweeper tool (see 5.1.1) and the parameters shown in table 5.2, an image pyramid is constructed for each image in the pair, then the features are detected using different values of the two configuration parameters. These features have descriptors computed with ORB, and then they are matched against the other image in the pair.

As in section 5.2, the number of successful matches is used to score the performance of the parameter configuration.

Figure 5.5 shows the matching performance for varying Harris matrix precision and Harris score precision. Beyond 16 bits of Harris matrix precision, no further improvement in matching accuracy is observed, irrespective of the score precision, so 16 is selected as `MATRIX_BITS`, the precision of *Tarsier*'s matrix calculations.

Figure 5.6 shows how changing the score precision affects the matching performance, holding the Harris matrix precision constant at 16 bits, for the same four image pairs as before. Beyond a precision of 24 bits, no improvement in matching performance is observed, so 24 is selected as the value for `SCORE_BITS`.

If memory is very tight, it may be worthwhile to reduce the precision allocated to these val-

Harris–Stephens	
Structure tensor window size	7×7
Non-maximum window size	3×3
ORB	
Quantisation angles	64
Moment window	37×37
Parallel windows	1
Matching	
Threshold	0
Prominence	20
Image Pyramid	
Octaves	2
Levels per octave	3

Table 5.2: Harris–Stephens analysis configuration

ues. In particular, the Harris–Stephens score precision shows a slower decline in performance as it is reduced, and so gains can be made here. At the same time, however, the Harris–Stephens score is only used in one 3×3 sliding window, so only two rows of scores need to be stored in memory at once. The matrix values, on the other hand, are stored in three 7×7 sliding windows, meaning that eighteen rows of the matrix values must be stored in memory at once. Thus, if memory pressure is high, reducing the precision of the Harris matrix scores will have the biggest impact on memory usage.

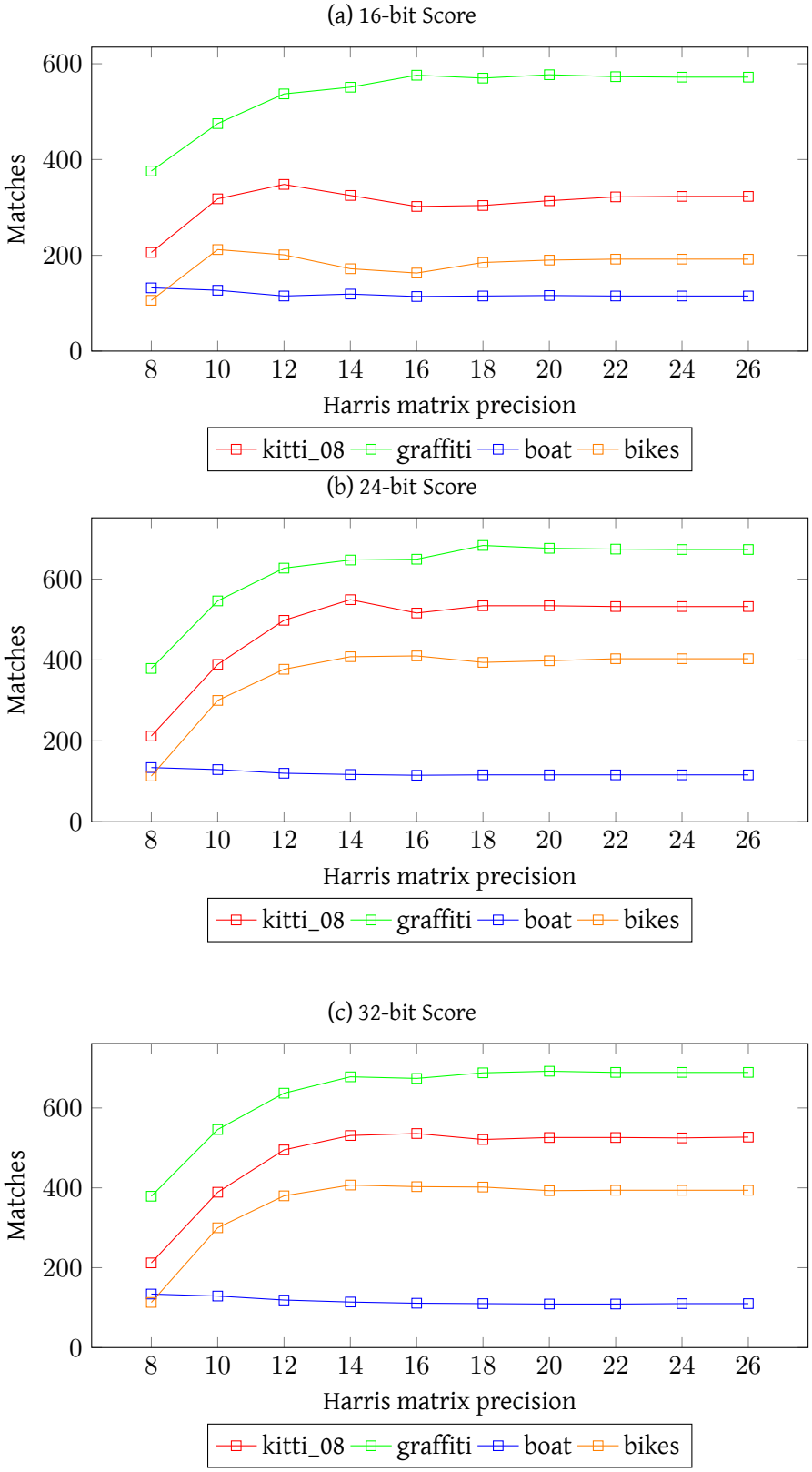


Figure 5.5: Match counts for the dataset images, varying the precision to which the Harris matrices are stored.

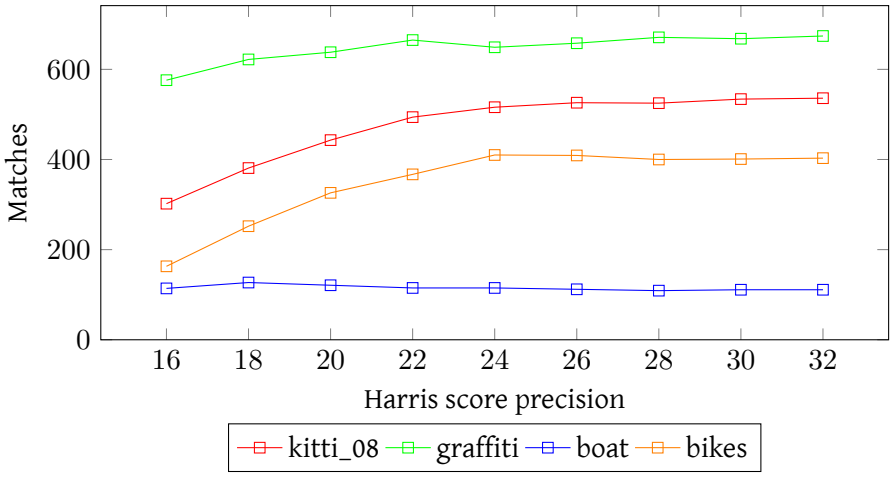


Figure 5.6: Match counts for the dataset images, varying the precision to which the Harris score is computed, while the Harris matrix precision is held constant at 16 bits

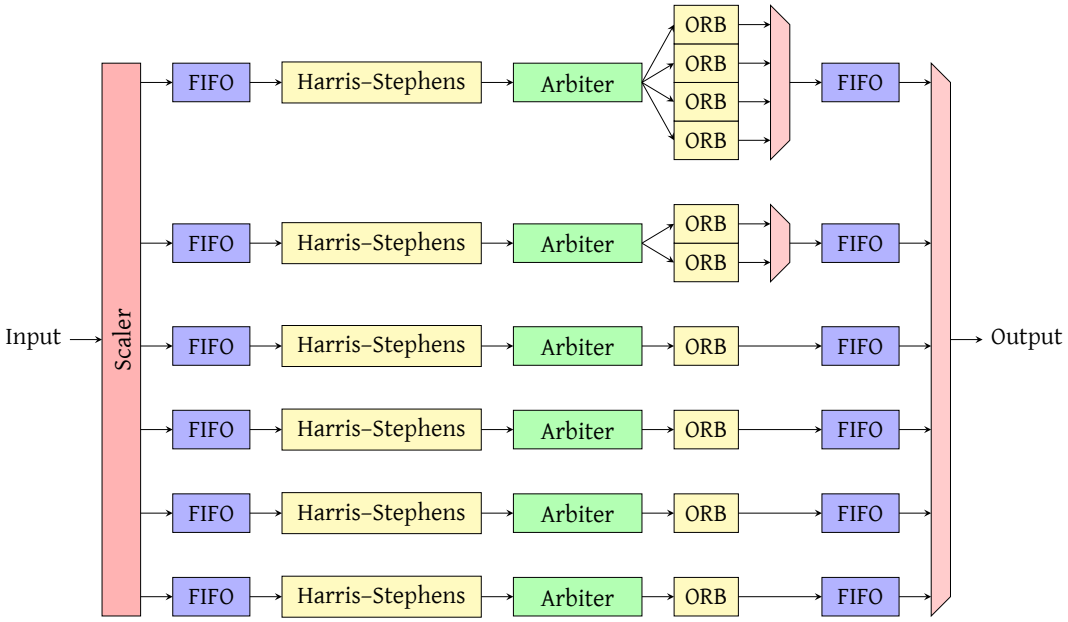


Figure 5.7: The multilevel feature extraction system [2] ©2017 IEEE

5.4 Feature Drop Analysis

5.4.1 Introduction

This section addresses the effects of different numbers of ORB modules at each level and different buffering strategies on the proportion of features that must be dropped by the feature extraction system. Three different buffering strategies, with two variants, are presented and their relative utilities compared using real images. The two images used present a contrast between low and high feature densities, for comparison.

Each level of the multilevel feature extraction system operates independently of all of the others. The input is a stream of pixels, and the output is a stream of features, in the form of coordinate pairs and 256-bit ORB descriptors.

The processing component of each level consists of a Harris–Stephens corner detector and one or more ORB feature extraction modules. As described in section 3.4.3, the ORB modules require 46 input pixel columns to fill their internal memories and then 266 clock cycles to generate the descriptor itself.

As shown in figure 5.8, each module has three states it can occupy — initial ***filling*** (unable to process, blue), ***ready*** (ready to process, green), and ***processing*** (unable to process further corners, red). When a corner is detected by the Harris–Stephens corner detector in each level, one ORB

processing module is selected from those in the ***ready*** state to process the incoming corner. This module then moves into the ***processing*** state. Once the descriptor is ready, the module moves into the ***filling*** state, where it remains until 46 consecutive input columns have filled its internal memory. The ***filling*** state is the initial state of all of the processing modules upon device reset.

As a special optimisation, if no input columns are received while the module is ***processing***, the ORB window remains up to date ("***clean***") and does not become inconsistent with the input ("***dirty***"). This means that the module can transition directly from ***processing*** back to ***ready***, saving the refill.

If no processing modules are available when a corner is detected, the corner must be ***dropped***, since some of the information needed to generate its descriptor is not present in the internal memory of any of the processing modules. In general, this is an undesirable event, as it means that that corner is unavailable for matching.

Since no input is required while a processing module is in the ***processing*** state, it is possible to buffer input pixels, potentially containing corners, in an input FIFO ahead of the corner detection system (see figure 5.7). This FIFO accepts input pixels from the scaling system and feeds them into the corner detector when requested. This means that the input into the level can be stalled, delaying the detection of features until a module is ready to process them.

This section deals solely with the input FIFO at the start of each level. The output FIFO is only responsible for buffering the output descriptors, to ensure that two levels do not attempt to output at the same time.

It is important to note that the modules must be refilled with input pixels before they are able to process a corner once more. Since stalling the input for a level stops input from reaching any of the modules in that level, no refilling progress can occur when input is stalled, only processing.

If input is being stalled because an ORB module is processing, and the FIFO becomes full, a problem arises when further input arrives. This input cannot simply be discarded, since that would cause the sliding window linebuffers to become desynchronised with the input, meaning that the feature detector and extractors would produce garbage thereafter. Instead, input must be forced into the detector/extractor chain, so that space is made in the FIFO for the newly-received input. This is termed an ***overrun***. The feature detector is capable of dealing with this input data, but corners detected while the buffer is in overrun may not have a module available to process them, and thus may have to be dropped.

The following analysis assumes that the host system supplies image data to the *Tarsier* at a constant rate of 1 pixel/clock. This data rate has been observed in the demonstrator, as the host system is reliably able to saturate the input buffers. If the host system does not sustain a 1 pixel/clock input rate, the feature drop distribution will be different, since the feature extractor modules can run while *Tarsier* input is stalled.

If the input data rate is 1 pixel/clock, the FIFO is unable to empty until the end of the frame, as the input rate is the same as the maximum output rate. This means that the fill level of the FIFO increases whenever input is stalled, with no way of reducing the fill level so long as input continues. This only arises in level 0 of the pyramid, but it means that this level is the most likely to overflow the FIFO buffer. At level 0, the FIFO will provide the ability to stall the input, reducing the drop rate, but only until it fills, and thus only for the first part of the image. Consequently, the use of buffering is of most use in higher levels of the pyramid, where it can reduce the number of ORB modules needed for a given drop rate.

This work investigates two potential strategies for stalling the FIFO input, based upon the states of the ORB processing modules, in addition to the control case, where no buffering is performed.

5.4.2 Guide to Timing Diagrams

Figures 5.10, 5.11, 5.12, 5.13 and 5.14 represent the timing of the states of two ORB modules for two basic sequences of pixels, selected because they show differences in strategy performance. Figure 5.9 shows a representation of parts of two input images. 56 pixels are shown for each image, with dark grey cells representing input pixels that are corners and white cells non-corners. The pixel stream enters the system from index 0 and continues until index 55, possibly being stalled by the choice of strategy in use. Both input images contain three corners, and the initial system state has both modules in the ***ready*** state and the FIFO empty.

All five diagrams assume unlimited buffer space and an input rate of 1 pixel/cycle, the same as level 0 of the pyramid. Section 5.4.7 addresses the situation in other levels of the pyramid, when the pixel rate is lower than the clock rate.

Figures 5.10, 5.11 and 5.12 show the operation of each of the strategies, without the use of the dirty bit, on these two pixel sequences. Along the top, the row labelled t shows time, measured in clock periods (this is not to scale). The next row down, labelled corner, displays a symbol if a

corner is detected at this point in time. Ticks indicate that a module is available to process the detected corner, and crosses indicate that no module is available, so the corner must be dropped.

The next two lines show the state of each module using colour. Green represents the ***ready*** state, in which all modules start in this steady-state analysis. Red represents the ***processing*** state, and blue shows the ***filling*** state. A hashed-out area indicates that the module would be filling, but the input is stalled, so it has to wait.

The third line, where present, indicates times where the input is stalled by the choice of strategy. Input stalls do not stall processing of descriptors, but they do delay refilling of modules.

Figures 5.13 and 5.14 show how the introduction of the dirty flag to the ORB modules affects performance. In these two diagrams, two additional rows are appended, showing the state of the ORB window's dirty flag. The flag is set (shown in grey) when input is received while the ORB module is in the ***processing*** state, meaning any time when a window is processing and input is not stalled. If the flag is set when the module finishes processing, it must refill its window before it can be marked ready again.

5.4.3 Strategy 0: No buffering

This strategy does not use the input FIFO buffer at all, and serves as a useful reference comparison against the other two strategies. Figure 5.10 shows a simulation of its operation. In both cases, the third corner to arrive is dropped since there are no modules available to process it.

Both modules start in the ***ready*** state. At $t = 0$, a corner is detected, and module 1 transitions into the ***processing*** state. At $t = 6$, a second corner is detected, so module 2 transitions into the ***processing*** state. At this point, neither module is available to accept detected features, and so when the third corner is detected, at $t = 54$ or $t = 50$, it must be dropped. The two modules complete processing and begin refilling, shown in blue, and eventually return to the ***ready*** state.

In general, the no buffering strategy has the lowest latency of the three methods — since the buffer is never used, there is minimal delay between the input of the final pixel in an Image And the completion of descriptor generation. Additionally, the memory requirements are lowest, since there is no buffer.

This method does however drop the largest number of corners of the three strategies.

5.4.4 Strategy 1: Stalling when all modules are processing

This strategy ensures that the input is stalled when there is no chance of a detected feature being processed. Whenever all the processing modules in a level are in the ***processing*** state, the input is stalled. When all the modules are processing, there is no use in feeding input, as any detected corners will necessarily be dropped, since there are no free modules. As soon as any one of the modules completes processing and moves into the ***filling*** state, input is resumed, allowing this module to be filled with pixels. Note that the requisite 46 columns still need to arrive before the module can move into the ***ready*** state, where it is able to process corners once more.

The same two examples used in the above section are shown in figure 5.11. The start is the same, but as soon as the second module beings processing, input is stalled, since both modules are unavailable. Once the first module begins refilling, input is resumed. The first example has the third corner arriving 48 pixels after the second, two pixels more than are required to refill completely, and so module 1 is available to process the third corner. The second example, however, has only 44 pixels between second and third corners, so it cannot be processed by either module and must be dropped. Although there are 50 pixels between the first corner arriving and the third, enough that the same module could theoretically handle both corners, input continues to flow until the second corner is detected, occupying both the modules and stalling the input.

5.4.5 Strategy 2: Stalling when any module is processing

This strategy is the most conservative of the three, ensuring that modules are returned to availability as soon as possible, so that as few corners as possible are dropped. It does this by stalling input whenever any module is in the ***processing*** state.

The primary justification for this approach is the refilling delay after processing. Since 46 columns of input are required after a successful feature extraction before the module is available to process another corner, this approach attempts to ensure that another module is available during the refilling phase.

This comes at a cost of heavy buffer usage — we buffer far more input data than the above strategy for a given input image. Thus it is only advantageous on higher pyramid levels, where the pixel rate is sufficiently low that the buffer can catch up to the input.

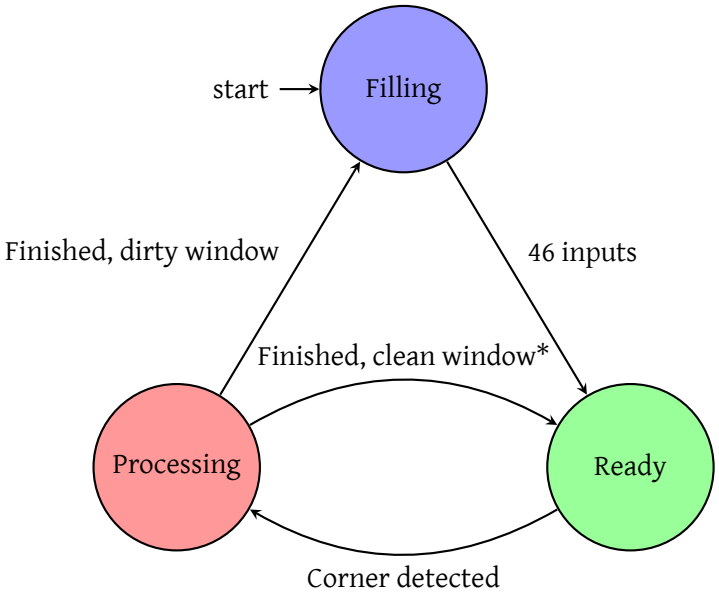


Figure 5.8: State machine for each ORB module. The "Finished, clean window" transition is only used when the dirty bit is enabled.

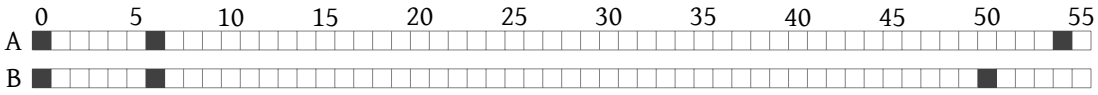


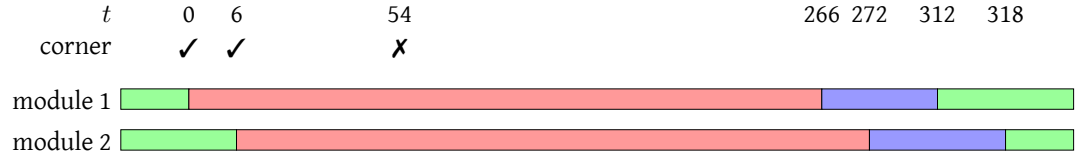
Figure 5.9: Two pixel arrival sequences. Each square represents a pixel entering the system, and black squares indicate pixels that are detected as corners. Sequence A has corners at time 0, 6 and 54. Sequence B has corners at time 0, 6, and 50.

The two examples are repeated with this strategy in figure 5.12. In both cases, the buffer is stalled immediately after the first corner arrives, and the second corner arrives once the first module is already in the ***filling*** state. This causes a further buffer stall, and again, input resumes once the second module is in the ***filling*** state. In both cases, the first module has returned to the ***ready*** state before the third corner arrives, meaning that all three corners are processed successfully.

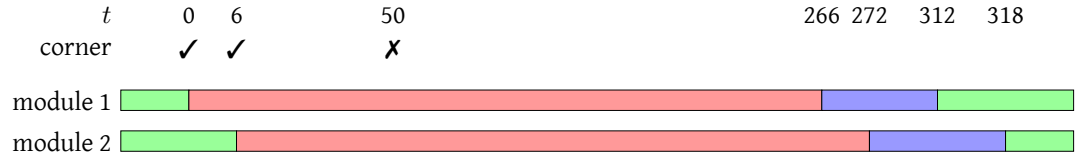
For one module, this strategy is identical to strategy 1.

5.4.6 Small-Scale Analysis

Figure 5.10 shows the strategy with no buffering of input, and hence no stalls. This approach yields the tightest real-time bounds on processing time, since the processing time is independent of the content of the image. As each feature is detected, it is delegated to the first available processing module. If not processing modules are available, the feature is dropped. Since the

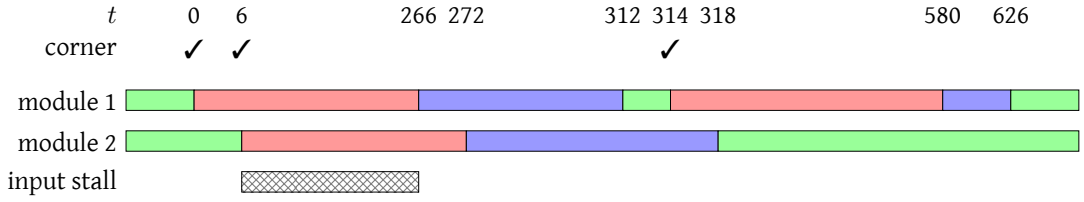


(a) Image A: Corners at 0, 6, 54

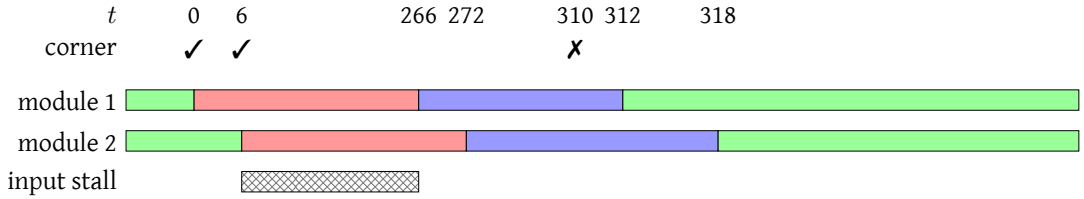


(b) Image B: Corners at 0, 6, 50

Figure 5.10: Use of two modules for the examples in figure 5.9 (Strategy 0 - no stalling)

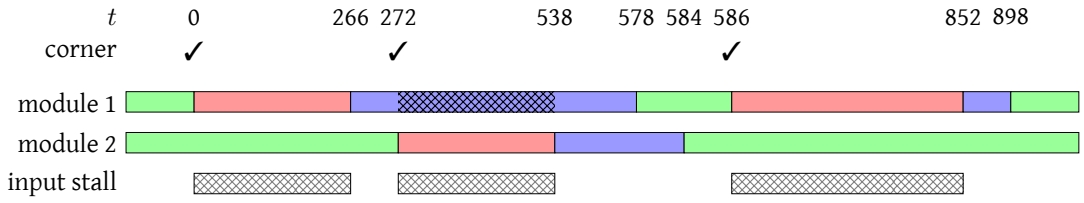


(a) Image A: Corners at 0, 6, 54

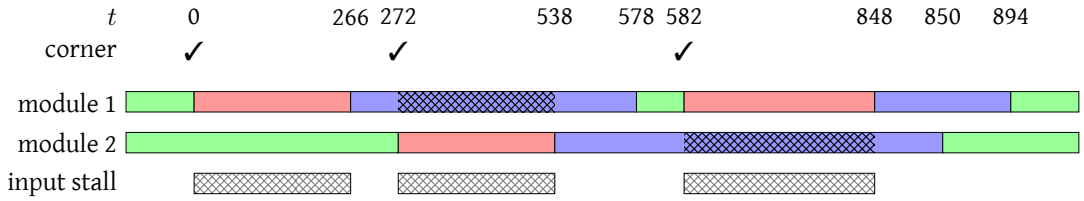


(b) Image B: Corners at 0, 6, 50

Figure 5.11: Use of two modules for the examples in figure 5.9 (Strategy 1 - stall when all modules processing)

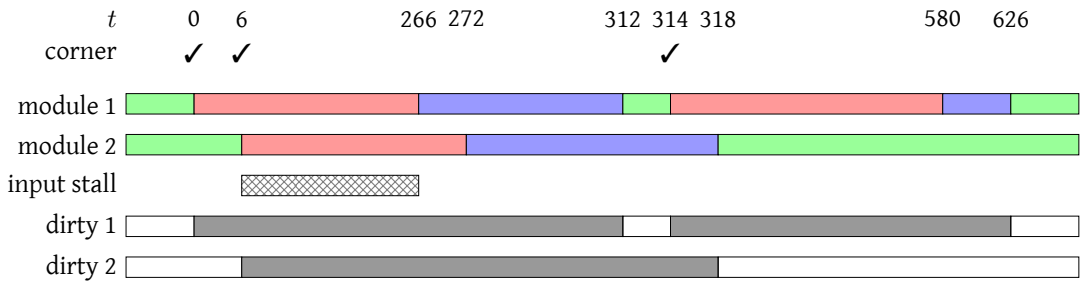


(a) Image A: Corners at 0, 6, 54

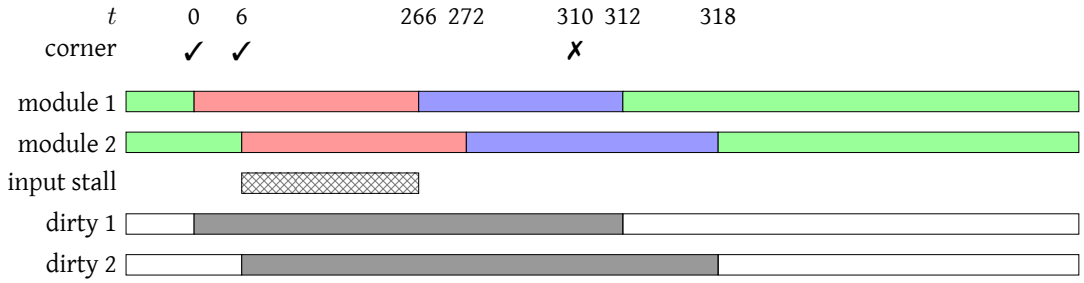


(b) Image B: Corners at 0, 6, 50

Figure 5.12: Use of two modules for the examples in figure 5.9 (Strategy 2 - stall when any module processing)

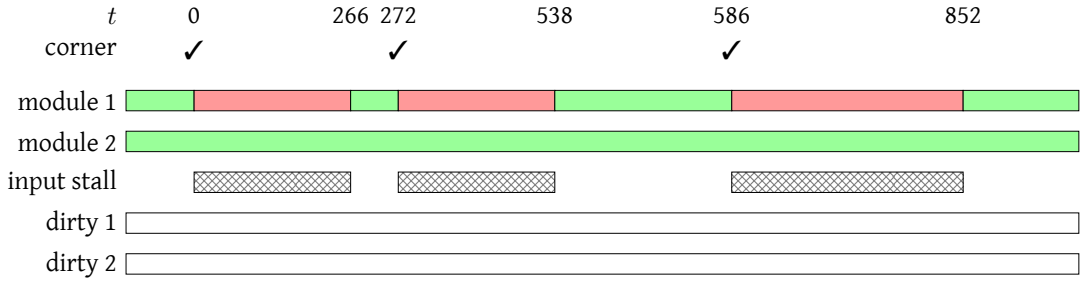


(a) Image A: Corners at 0, 6, 54

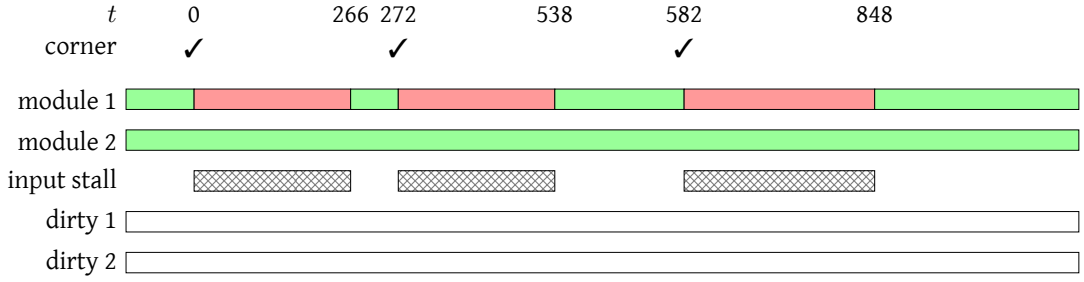


(b) Image B: Corners at 0, 6, 50

Figure 5.13: Use of two modules for the examples in figure 5.9 (Strategy 1 - stall when all modules processing, dirty bit in use)



(a) Image A: Corners at 0, 6, 54



(b) Image B: Corners at 0, 6, 50

Figure 5.14: Use of two modules for the examples in figure 5.9 (Strategy 2 - stall when any module processing, dirty bit in use)

processing time for a corner is much more than the interval between the detected features, this strategy fails to process the third corner in either of the two sequences presented in 5.9.

Figure 5.11 shows the strategy where input is blocked if all modules are processing. This halts input in particularly dense regions of the image, so that fewer features are lost, compared to no buffering. The two pixel sequences are selected to show a difference with this buffering mode. The strategy handles sequence A well, since module 1 become ready again immediately before the detection of the feature in pixel 54. Sequence B, on the other hand, is not handled as well. The feature in pixel 50 arrives while both modules are refilling, and thus the feature is dropped. Input cannot be stalled while both are refilling, since they would never be able to finish. Notice that although the strategy handles sequence A successfully, the time taken to do so is significantly longer than strategy 0, since the input is blocked for 260 clock cycles.

Figure 5.12 shows the strategy where input is blocked if any module is processing. Input is stalled as early as possible, in order to ensure that as few modules as possible are unavailable due to needing to refill. This comes at the cost of increased total processing time, since the input is stalled at times when it might not be necessary to attain minimum drop rate. This strategy successfully processes all three corners in both sequence A and sequence B. In both sequences, the first corner is processed on module 1, and then the second is processed on module 2 while module 1 is refilling. Processing the second feature causes the input to be stalled once again, meaning that the refill of module 1 takes significantly longer than other refills (312 clock cycles instead of 46). In both sequences, at least one module is available to process the third feature to be detected. It is still possible for the strategy to drop a feature, since it is possible for all modules to be refilling when a feature is detected, but this will happen less frequently than with strategy 0 or strategy 1. Since this strategy makes such heavy use of the FIFO buffer, it is better suited to higher pyramid levels, where the decreased pixel rate allows more chance to catch up with the input by emptying the FIFO.

Figure 5.13 shows strategy 1 again, but this time with the use of the dirty flag. The dirty flag can boost the efficiency of modules by allowing them to return immediately to readiness without refilling, so long as no input has been missed. The dirty flag is ineffectual without the ability to stall the input, since input will always be received while the feature is being detected[4]. It is also

[4]This is because of the margin region at the end of each row of the image. A feature cannot be detected less than 18 pixels before the end of the row, and even at the highest level of the pyramid, at least one more pixel will arrive before the 266-cycle descriptor processing is complete.

ineffectual when used with more than one ORB module. If input is not stalled when one module is processing, as in strategy 1, the input supplied to the other modules will immediately cause the first window to be marked dirty, removing the advantage of quick return to availability. If input is stalled when one module is processing, as in strategy 2, that module will return to availability immediately upon finishing processing, as shown in figure 5.14. This means that unless the FIFO fills up and starts forcing input through, any further modules will never be pressed into service, and will always sit idle. In this case, further modules will have a benefit, processing features that would otherwise be dropped.

Use of the dirty flag with a single module is usually the most hardware-efficient approach. So long as the FIFO does not fill, a system using the dirty flag and a single module will never drop a corner. This dropping performance comes at the cost of time performance, but because the time taken to refill the windows is avoided, total image processing time is less than strategy 2 without the dirty flag.

5.4.7 Large-Scale Analysis

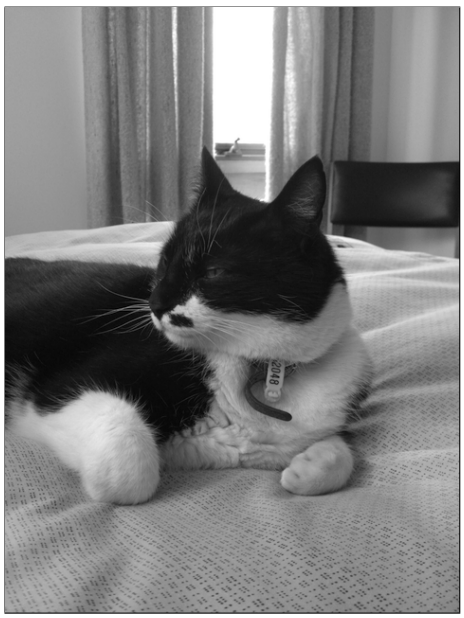
This section investigates the use of different strategies at different levels of the pyramid and their effects on feature drop rates. This analysis is performed by taking two images (see figure 5.15), constructing an image pyramid for them, detecting all the features in the image pyramids, then feeding the detected features into a simulation of the module timing, using the Feature Drop Simulator described in 5.1.1. The settings for the image pyramid and the feature detector are shown in table 5.3.

The two images have been selected to display different corner densities. The high-contrast text of Newton's Principiæ has a very high density of corners, whereas the photo of a cat on a bed has a much lower density. Since the density of corners is what most affects drop performance, the two images provide a useful contrast. Some basic statistics for the corner detection are shown in table 5.4.

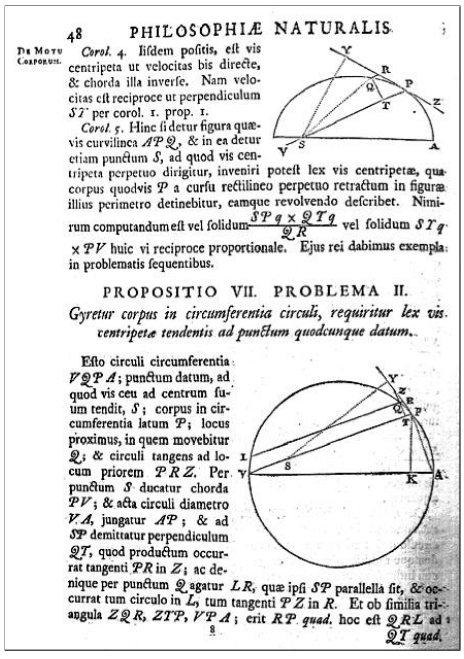
Since processing a feature takes 266 clock cycles, a single module will be unable to process all of the features in a level if the feature temporal density is less than 1/266, unless the entire image may be buffered. ***Temporal density*** is defined to be the the reciprocal of the average number of clock cycles between successive feature detections, or equivalently, the probability that a feature will be detected in a given clock cycle. Feature detection approximates a Poisson process, where

Harris–Stephens	
Matrix bits	16
Score bits	24
Structure tensor window size	7×7
Non-maximum window size	3×3
Threshold	200
ORB	
Quantisation angles	64
Moment window	37×37
Parallel windows	1
Matching	
Threshold	0
Prominence	20
Image Pyramid	
Interpolation	Bilinear
Octaves	2
Levels per octave	3

Table 5.3: Drop analysis configuration



(a) cat.png



(b) principia.png

Figure 5.15: The two images used in this analysis

Level	Corners	Spatial density	Temporal density
0	1073	1/286	1/286
1	454	1/433	1/677
2	182	1/690	1/1688
3	143	1/537	1/2148
4	106	1/464	1/2898
5	78	1/402	1/3938

(a) Corners detected in cat.png (480×640)

Level	Corners	Spatial density	Temporal density
0	2386	1/116	1/116
1	1759	1/100	1/157
2	1149	1/98	1/240
3	733	1/94	1/376
4	425	1/103	1/649
5	287	1/98	1/960

(b) Corners detected in principia.png (441×625)

Table 5.4: Corner statistics from two images. Spatial density is the reciprocal of the average number of pixels processed per feature detected, and temporal density is the reciprocal of the average number of clock cycles between successive feature detections.

the probability of detecting a feature in a given length of time is constant for a particular image level. There are some deviations from the ideal Poisson process, however, since two features cannot be detected in consecutive pixels (thanks to the non-maximum suppression), and since features are not evenly distributed.

Figures 5.16, 5.17, 5.18, 5.19 and 5.20 show the raw counts of features successfully processed at several levels of each pyramid, using varying numbers of modules and strategies. For each of the two sample images, a timing simulation, using the Rust feature drop simulator, is run for 1 to 5 modules and using the three stalling strategies described above. For the two strategies that actually use the FIFO, the effect of enabling the dirty flag, and only flushing the ORB windows when necessary, is shown as well.

Level 0 of the pyramid is a unique case (see figure 5.16). Since no scaling is performed on its input, the input rate is a constant 1 pixel/cycle. This means that the module can never empty its FIFO until it reaches the end of the image, since it cannot extract pixels from the buffer any faster than they are inserted. This means that the buffer rapidly fills up and then provides no more use, and as a consequence, every buffering strategy performs essentially identically for both images. Increasing the number of modules, however, has a significant effect on the drop performance. With every additional module, the number of features drops significantly. The

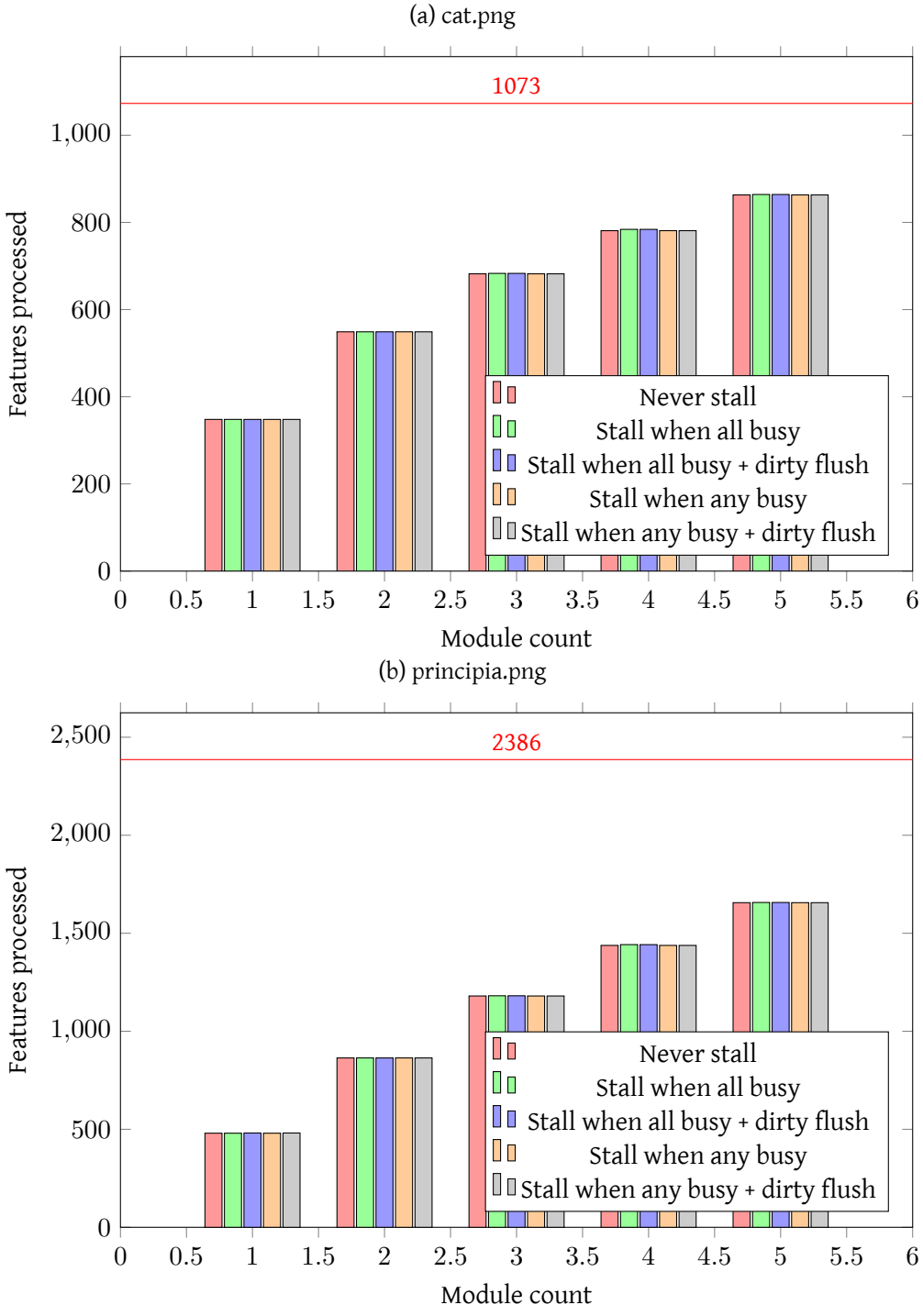


Figure 5.16: Raw count of features processed at level 0 of the two images, with a 512-pixel FIFO. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

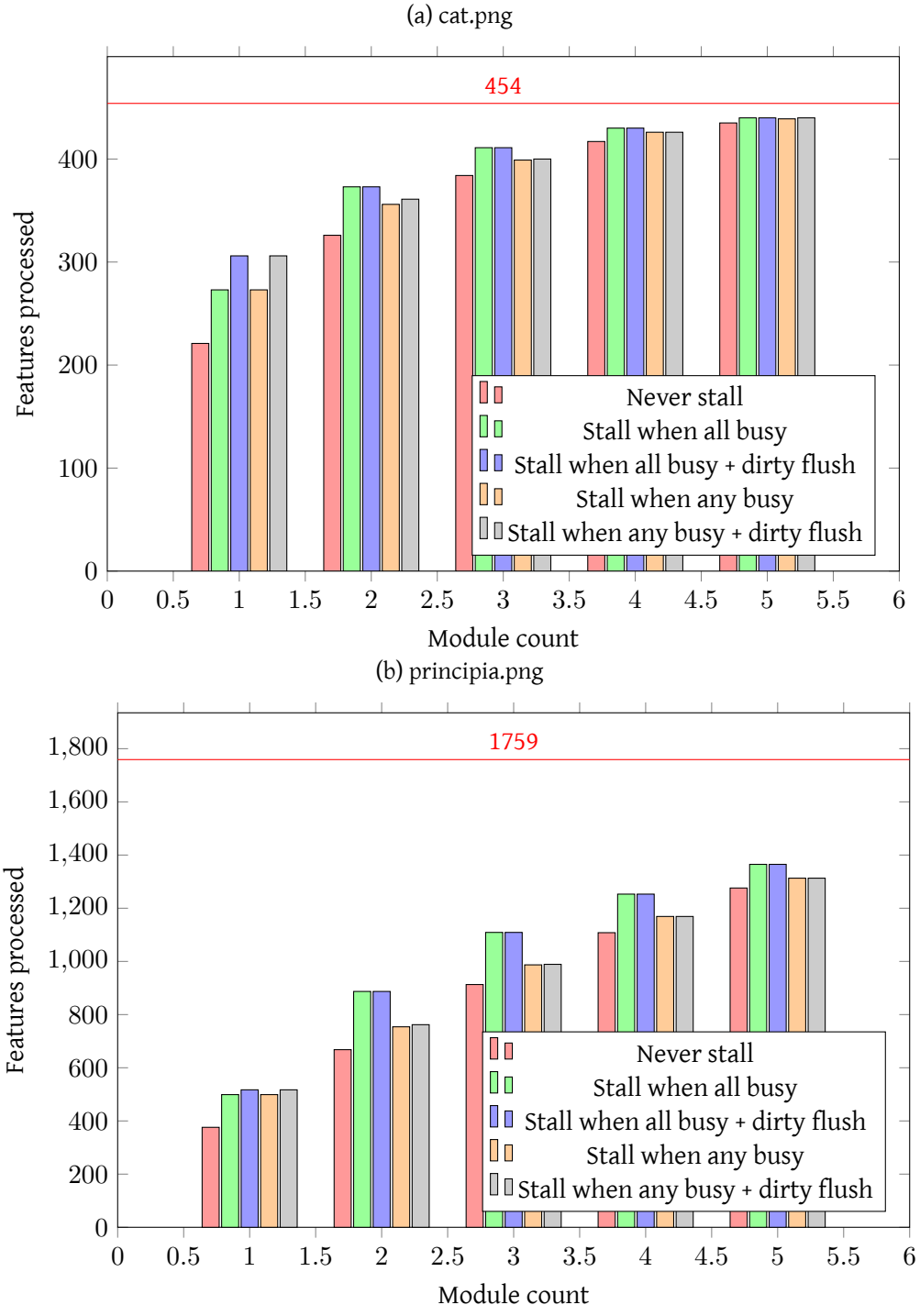


Figure 5.17: Raw count of features processed at level 1 of the two images, with a 512-pixel FIFO. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

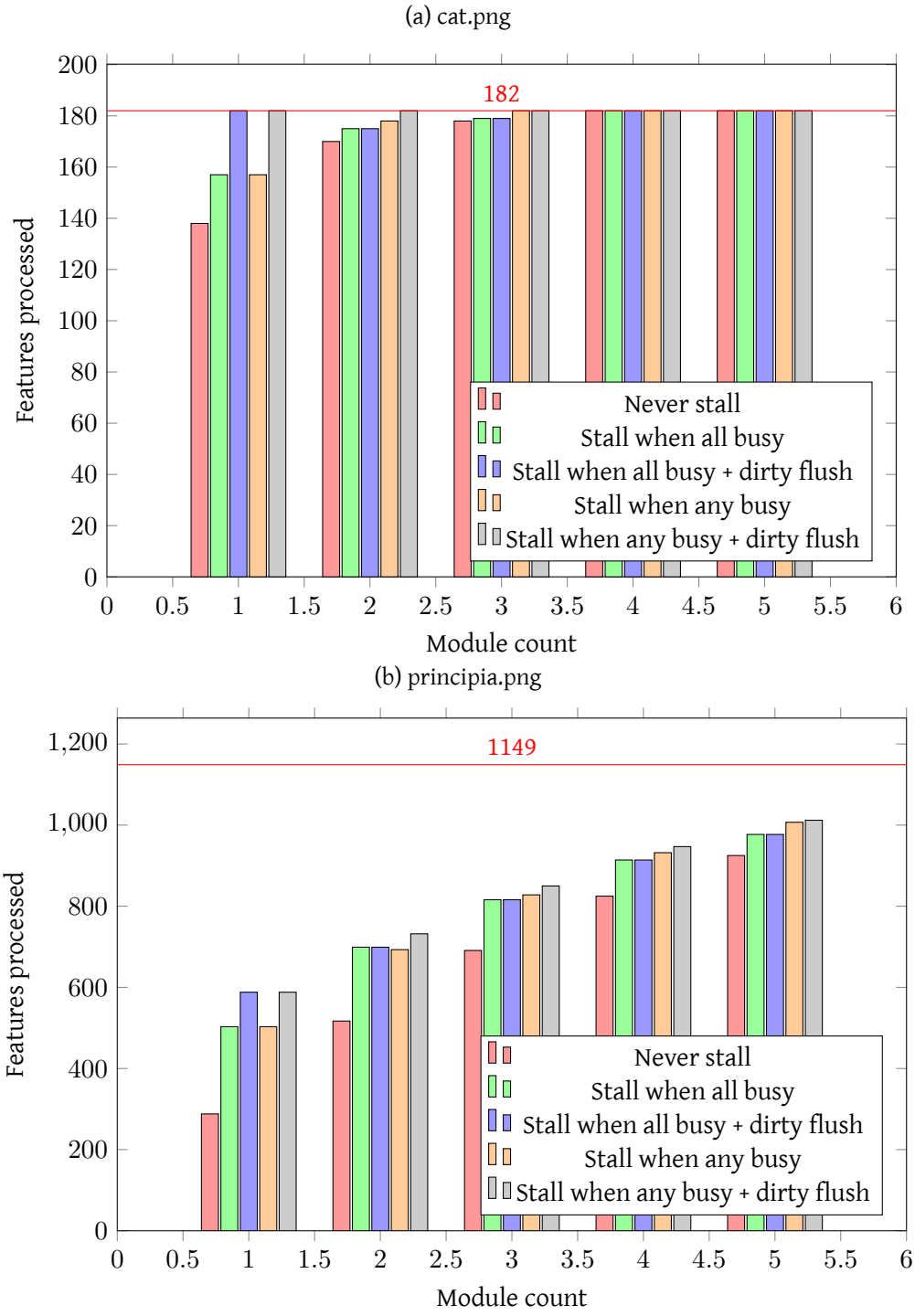


Figure 5.18: Raw count of features processed at level 2 of the two images, with a 512-pixel FIFO. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

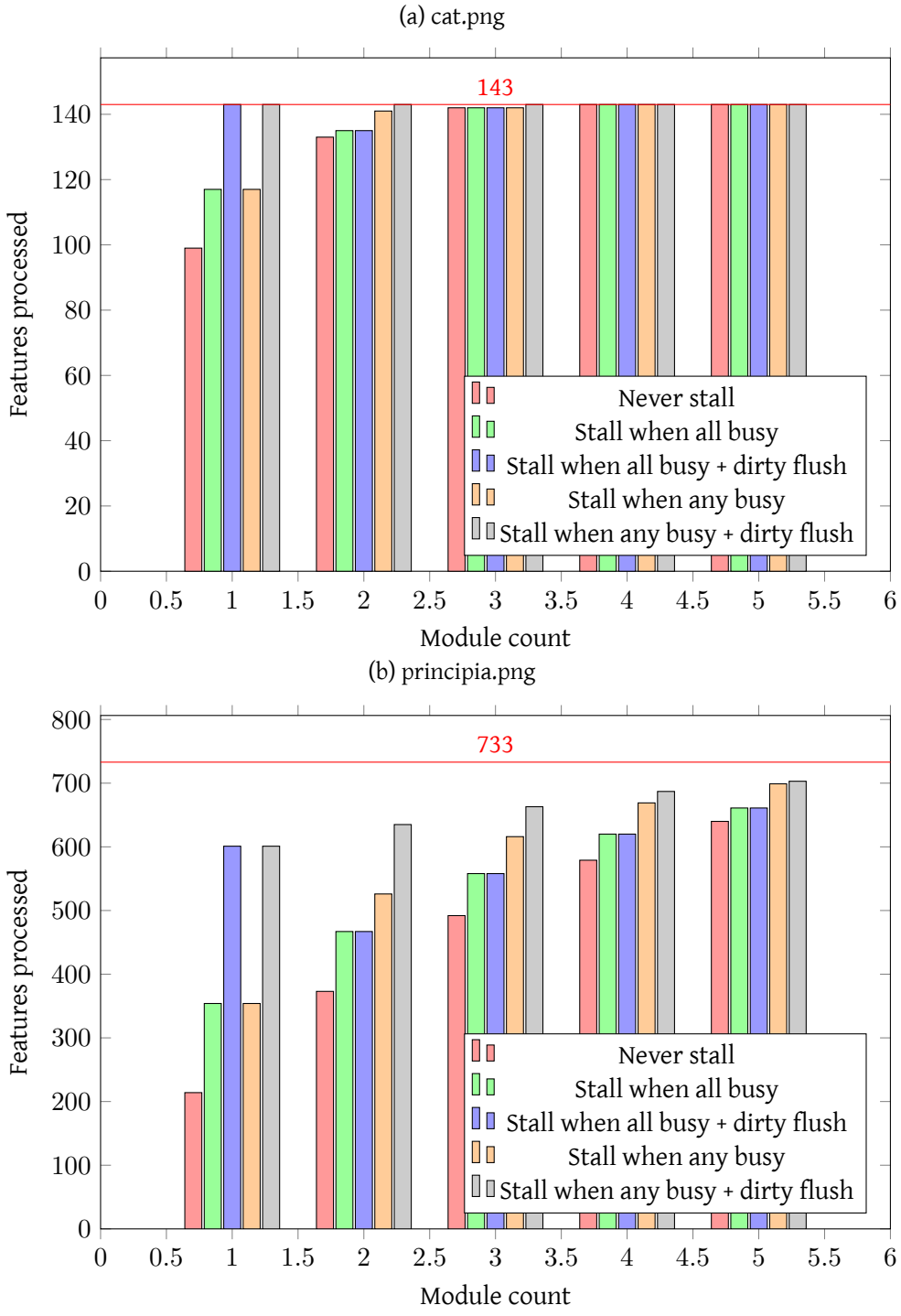


Figure 5.19: Raw count of features processed at level 3 of the two images, with a 512-pixel FIFO. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

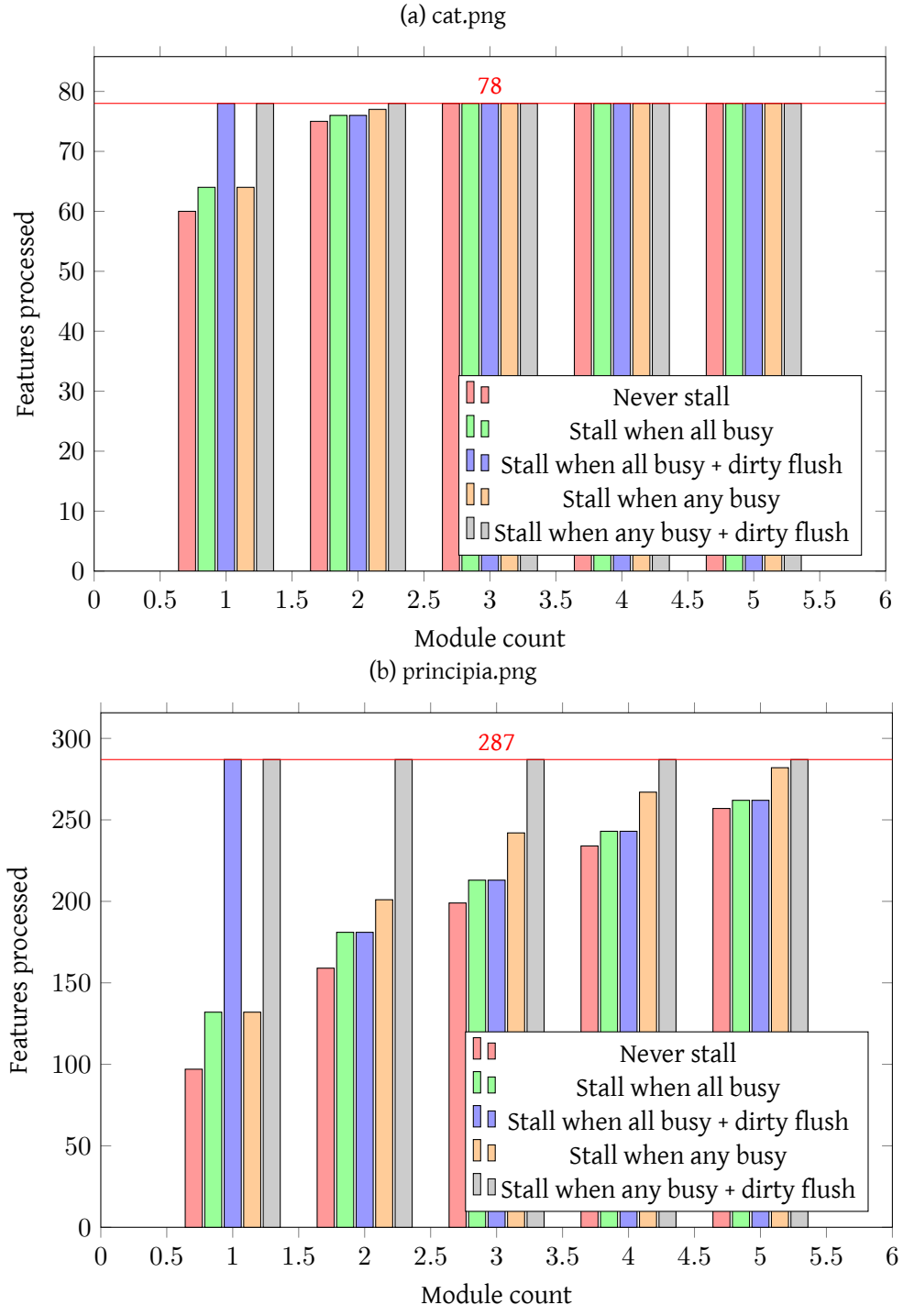


Figure 5.20: Raw count of features processed at level 5 of the two images, with a 512-pixel FIFO. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

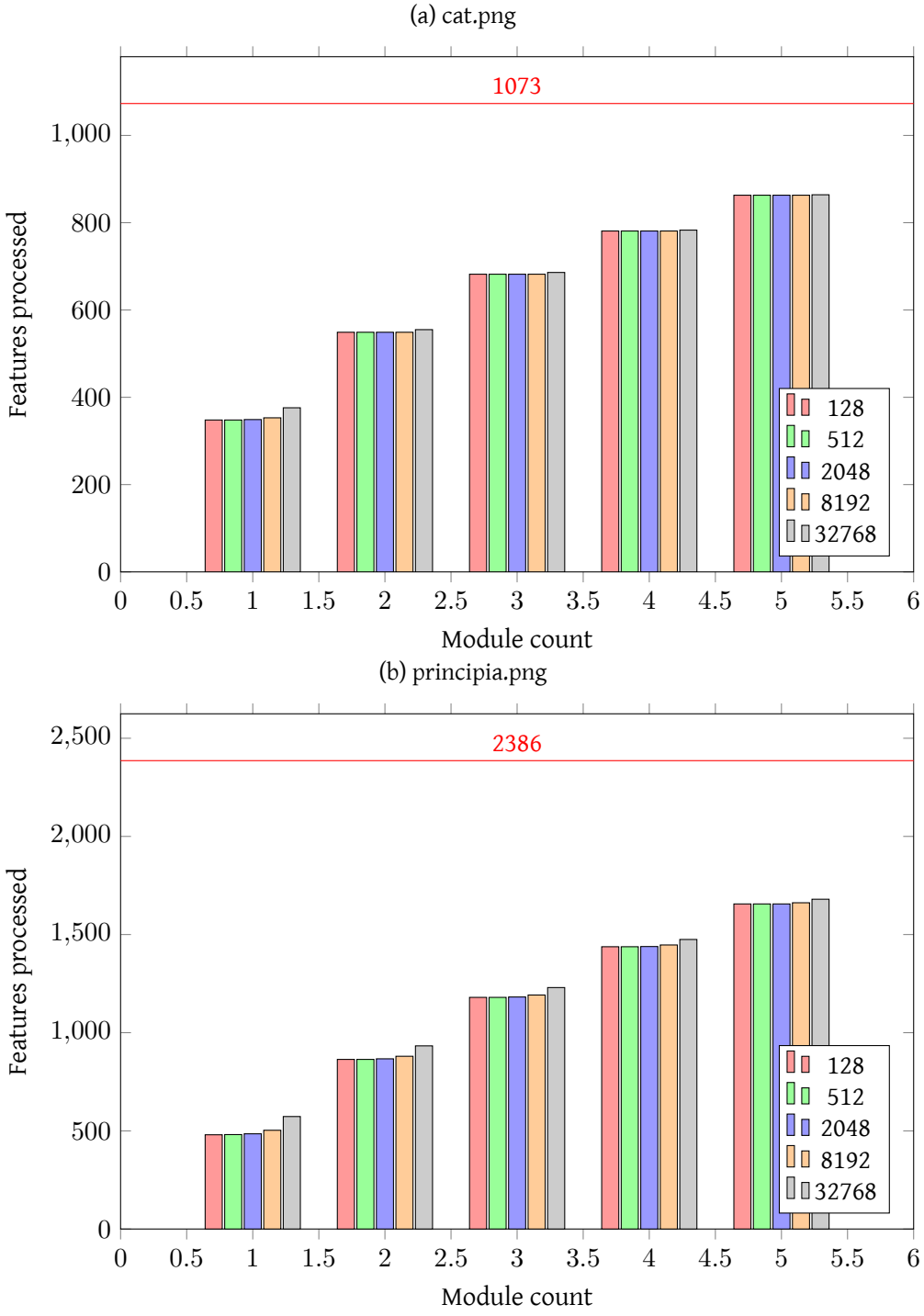


Figure 5.21: Raw processed feature count at level 0 using strategy 2, varying number of modules and FIFO length. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

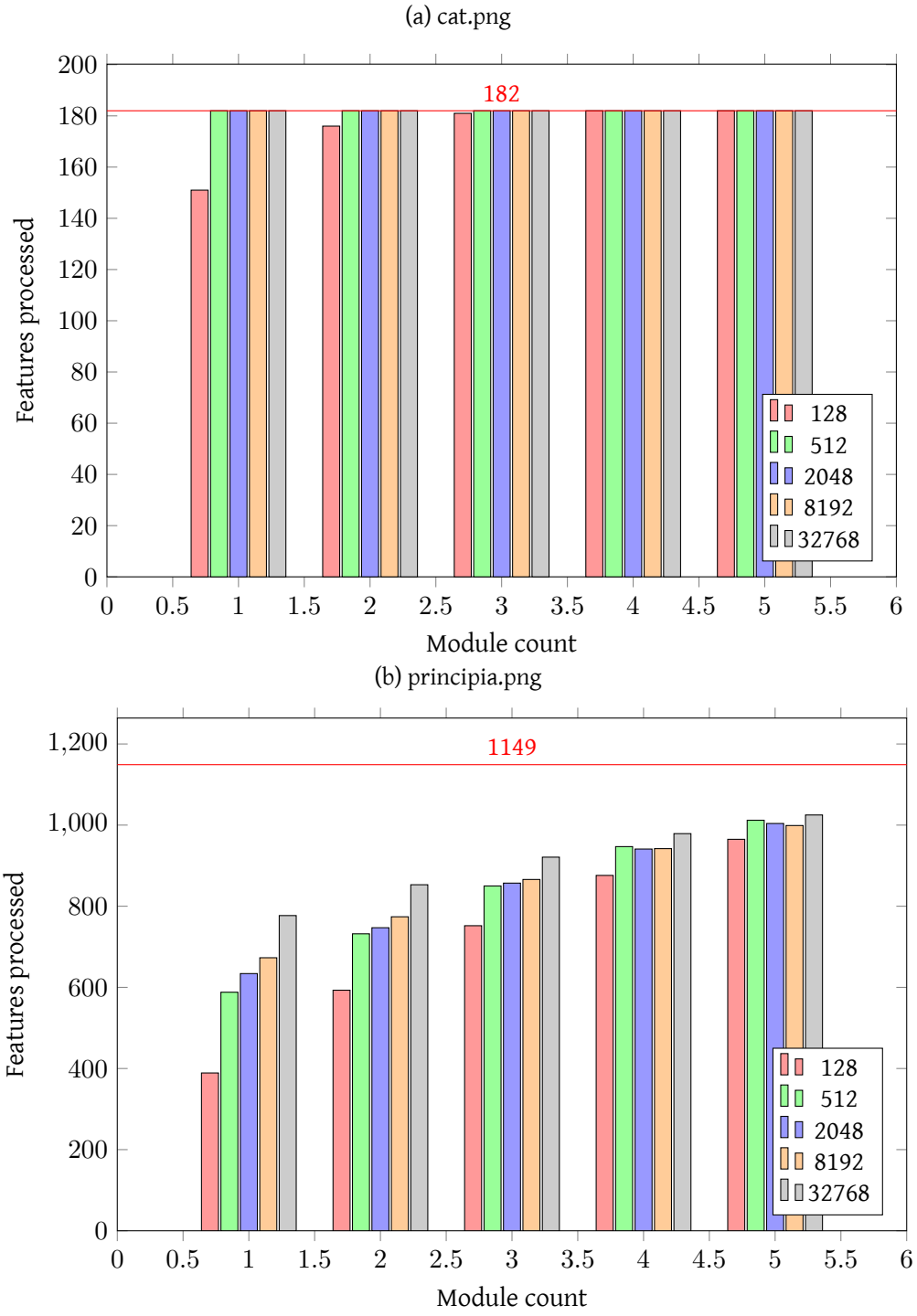


Figure 5.22: Raw processed feature count at level 2 using strategy 2, varying number of modules and FIFO length. The number of features in the image is shown as a red line. This represents dropping no features, and is the target for the system.

effect grows smaller with each additional module added, with 8 % more features processed with five modules compared to four.

Level 1 of the pyramid exhibits a curious behaviour (see figure 5.17. This level is able to take advantage of the buffering, since its input pixel rate is around 0.64 pixels/clock cycle. Strategy 1 (stall when all busy + dirty flush) has the best performance for all module counts, although the dirty flag has no effect for more than one ORB module. The margin over strategy 2 (stall when any busy + dirty flush) is small, but significant. This is due to the more effective use of multiple modules in parallel achieved by strategy 1. Since input continues to flow while one module is processing, fewer input stalls are required. If no other features are detected until the first module has finished processing and refilling, no buffer space is used, and no buffer overruns can occur. The timing simulation shows many more overruns on strategy 2 compared to strategy 1, as many as five times as many for the case with five modules. At the particular pixel rate observed in level 1, the tradeoff between strategy 1 and strategy 2 is in favour of strategy 1.

By comparison, in levels 2, 3, 5, and beyond, strategy 2 is the clear victor in dropping performance, especially when using the dirty bit. Since the pixel rate is yet lower, the time taken to process a feature represents fewer pixels entering the system than at lower levels of the pyramid. This means that the cost of stalling the input is small, since the detector/extractor chain can rapidly catch up with its buffer. For the cat image, a single ORB module, using strategy 2 and the dirty flag, is sufficient to process every single feature in levels 2 and up. For the text image, with its extra feature density, only 52% of the features are processed successfully at level 2, but the rate is 82% at level 3 and 100% thereafter. The temporal feature density at level 2 of principia.png is $\frac{1}{240}$, meaning that the average time between features is less than the time taken to process them, and so it is impossible for a single module to ever process all of the features. If the hardware is available, it might be advantageous to use an additional module at level 2, but otherwise one module should be sufficient for levels 2 and above.

Figures 5.21 and 5.22 show how drop performance is affected by FIFO length at levels 0 and 2. Extreme FIFO sizes at level 0 show a modest improvement in drop performance, but this is most likely due to features being detected while the buffer is still filling at the top of the image. A buffer of size 32768 represents $32768 \div 480 \approx 68$ rows of image data, or about 10% of the image, so for the first 10% of the image, the buffer will allow the modules to stall input. After this point, the FIFO will be full, and no further stalling will be possible.

The effect of larger buffer sizes is more interesting at level 2. A reduced buffer size of 128, down from 512, causes a major performance handicap at this level, providing significantly worse drop rate than 512. Increasing the buffer size beyond 512 does yield slightly more features processed, but the effects are small. A buffer size of 32768 does show a more pronounced improvement, most likely for the same reasons as for level 0.

These results suggest that a buffer of size 512 is an appropriate choice, with a small increase in size offering modest improvements in drop rate. If the hardware is available, this may be a valid selection.

5.4.8 Conclusion

In summary, the best choice of buffering strategy is strategy 1 at levels 0 and 1 and strategy 2 at levels 2 and up, using the dirty flag in all cases. A good strategy selection gives a boost in performance for negligible hardware cost, and so it is a clear recommendation.

A buffer size of 512 is appropriate for all levels. Less than this and performance suffers, more than this and the extra memory is wasted, because the performance gains are so small.

The number of modules to select at each level is a more complicated decision, and it depends very much on the resources available and the number of levels desired. The system as implemented has enough memory for 10 modules, and uses the arrangement

Level	Modules
0	4
1	2
2	1
3	1
4	1
5	1

This uses the largest number of modules where they stand to make the most improvement, specifically at levels 0 and 1.

5.5 Testbenching

Accompanying each of the functional modules of the SystemVerilog design is a testbench script, also written in SystemVerilog, whose role is to verify the correct functioning of the module. Many of these testbenches will feed the module a basic image and check that the result matches what is calculated by the Rust simulation utilities. There are several additional utilities that will take an image, detect its features and compute their descriptors, then prepare the image for input through a testbench into the module.

The testbenches are run in ModelSim[5], and it is confirmed that the results from ModelSim match the simulation results. This double-checking process provides additional confidence that the hardware has been written successfully. The testbenches have been an invaluable tool in debugging the design and validating changes.

[5] `https://www.altera.com/products/design-software/model---simulation/modelsim-altera-software.html`

5.6 *Tarsier*: Integrating Corner Detection, ORB Feature Extraction, Scaling and Buffering

The *Tarsier* device provides a wrapper around the feature detection and extraction system, exposing it as a hardware coprocessor to a computer system. The reference *Tarsier* implementation is constructed on an Altera Arria V Starter Kit board and connects to a desktop PC via PCI Express. Both input and output are carried over a PCI-e 2.0 x4 connection, providing 2 GB/s of bandwidth in each direction.

5.6.1 Input

The input to the *Tarsier* is an uncompressed bitmap image.

Input to the *Tarsier* is handled by a Scatter-Gather DMA controller. This controller reads data from host system memory then streams it to the feature detector. PCI Express grants access to the host system bus, so the DMA controller is able to read without the involvement of the host CPU.

Although the Arria V Starter Kit board has an ethernet socket, suitable for connecting an ethernet camera, PCI Express was chosen as the input interface, as it allows more flexibility in the source format. Where an ethernet camera might have allowed lower latency, taking the images directly from the camera, it does not allow for images to be taken from a prerecorded source instead of a live camera. A prerecorded source makes it simpler to achieve reproducible results, which is important in testing.

The Scatter-Gather DMA controller allows multiple short transfers to be queued sequentially, which are then processed in order. This allows images to be stored in memory with padding between rows, as each row can be queued as a separate transfer.

Each frame is prefaced with a short header containing its dimensions and the desired Harris–Stephens threshold, and the frame pixels themselves are transferred in 8-bit greyscale, raster scanned left to right, top to bottom.

The maximum rate that the *Tarsier* device can accept image data is constrained by its core clock speed. The implemented frequency is 150 MHz, and it can accept one 8-bit pixel per clock cycle, thus the maximum data rate is 150 MB/s, well below the interface maximum.

5.6.2 Output

The output from the *Tarsier* is an array of detected features.

The output from the *Tarsier* device is transferred into system memory using another DMA controller. The output is in the form of an array of structures, with each structure representing one detected feature. The structure for each feature contains the 256-bit ORB descriptor, its x- and y-coordinates, and the pyramid level on which it was detected, with the total structure size being 40 bytes.

Once all the features detected in an image have been copied into system memory, the *Tarsier* device signals an interrupt on the host system. This is handled by the driver, which then passes the feature array to the user mode program making use of the accelerator.

Since it it is possible for an ORB feature extraction module to process one descriptor every 266 clock cycles, and there are 10 modules on the *Tarsier* device, it is possible for one 40-byte output feature to be emitted every 26 clock cycles at peak burst rate. For this reason, the output data path is 32 bits wide, allowing each feature to be transmitted to the host PC in 10 cycles, and ensuring that the output rate is never the bottleneck.

5.6.3 Integration

This work employs the *Tarsier* as an accelerator for the ORB-SLAM2 software SLAM system developed by Mur-Artal et al[5]. ORB-SLAM2 runs a customised version of the ORB code from OpenCV[6] on CPU, without taking advantage of any hardware acceleration. Feature detection is performed using FAST (see 2.2.2), and corners are selected from images using an octree, with the goal of ensuring relatively even distribution across the image.

ORB-SLAM2 has been modified slightly to allow it to make use of the *Tarsier* accelerator. It has been refactored to allow multiple feature detection/extraction backends to be selected at runtime—the original software extractor, the *Tarsier* hardware, and a software reference implementation of the *Tarsier*. Subpixel refinement in ORB-SLAM2 has had to be disabled, since it required access to the raw images from the downsampling pyramid, which are no longer available. They are generated on demand on the *Tarsier*, and never appear as a whole image, let alone in system memory.

[6] `https://opencv.org`

Parameter	Value
Maximum width	2048
Maximum height	2160
Harris–Stephens	
Matrix bits	16
Structure tensor window size	7×7
Score bits	33
Non-maximum window size	3×3
ORB	
Quantisation angles	64
Moment window	37×37
Parallel windows	1
Image Pyramid	
Resampling method	Bilinear
Octaves	2
Levels per octave	3
Buffer length	512
Level 0 modules	4
Level 1 modules	2
Level 2 modules	1
Level 3 modules	1
Level 4 modules	1
Level 5 modules	1
Level 0 strategy	Stall when all busy + dirty clear
Level 1 strategy	Stall when all busy + dirty clear
Level 2 strategy	Stall when any busy + dirty clear
Level 3 strategy	Stall when any busy + dirty clear
Level 4 strategy	Stall when any busy + dirty clear
Level 5 strategy	Stall when any busy + dirty clear

Table 5.5: The selected configuration parameters for the *Tarsier* device

All three backends still make use of the existing feature matching and tracking code in ORB-SLAM2.

5.6.4 Performance

Table 5.5 lists the values selected for each of the configurable parameters in the system.

Table 5.6 shows the resource utilisation, calculated F_{max} and estimated power consumption of the compiled *Tarsier* device.

Table 5.7 shows the amount of each resource various subdivisions of the *Tarsier* use. The memory consumption of Harris–Stephens depends on the maximum width of the image, and so it is given as a range.

With these compilation parameters, the *Tarsier* is able to process input pixels at 150 MPix/s,

	Value	%
Logic elements (ALMs)	103,605	76%
Memory bits	7,328,480	41%
Multipliers	272	26%
F_{max}(fast model)	150 MHz	
Power	4778 mW	

Table 5.6: Resource utilisation, calculated F_{max} and estimated power consumption of the compiled *Tarsier* device

Module	ALUTs	Registers	Memory bits	Multipliers
PCI-express	10953	10681	307232	0
Feature extractor	128421	250162	7021248	272
Control logic	2171	3170	4608	0
Level 0	32017	76842	1573408	97
Level 5	17776	30855	786248	28
Harris–Stephens	11492	13625	442368	5
ORB module	4782	15673	21320	23

Table 5.7: Resource utilisation by module

and features at a typical rate of 170 k features/s. This pixel rate is equivalent to 72 fps for 1920×1080 (Full-HD) images, and 488 fps for 640×480 images. A quad-core Intel 2500K CPU running at 4.3 GHz running the OpenCV implementation of FAST and ORB manages 48 fps for 1920×1080 images, demonstrating the clear performance advantage of the *Tarsier* device. The CPU has a rated Thermal Design Power (***TDP***) of 95 W, and, although it is challenging to measure, the CPU was probably drawing at least half of this value when running all four cores.

The CPU implementation does not allow parallelism over an image. Dividing an image into smaller tiles and processing each tile with a separate thread would result in a reduced latency and increased throughput, but such an operation was not supported using the tested API. Instead, parallelism was implemented by processing multiple images simultaneously, starting each image as soon as it was received from the input stream. With this approach, each image takes the same length of time to process, but more can be processed in the same time, compared with a single CPU thread.

A GPU, specifically an Nvidia GeForce GTX 970, was also used as a comparison, using OpenCV's OpenCL implementation of FAST and ORB (there is no GPU-accelerated Harris–Stephens algorithm in OpenCV). This device was only able to process 1920×1080 frames at 19 fps, and has a much higher TDP than the CPU, over 200 W.

Table 5.8 shows the relative performances for the three devices. The performance figure

Device	Throughput	Frame Latency	1080p FPS	Energy per frame (approx)
CPU (1 thread)	28 MPix/s	74.1 ms	13.5	405 mJ
CPU (4 threads)	100 MPix/s	74.1 ms	48.2	430 mJ
GPU	40 MPix/s	52.1 ms	19.2	15.5 J
Tarsier (fast model)	150 MPix/s	13.8 ms	72.3	74 mJ

Table 5.8: Performance comparison of the *Tarsier* device against a CPU and a GPU, for 1920×1080 frames. The 4-threaded CPU implementation assigns frames to a specific thread, processing multiple frames in parallel, but not reducing the per-frame latency. [2] ©2017 IEEE

shown for the ***Tarsier*** is determined using the *Fast Model*. This is the speed achievable using a mid-range silicon bin, that is an FPGA with low electrical impedance. The achievable clock speed on a low-quality silicon FPGA will be lower.

5.6.5 Testing

The integrated system was tested by running images from the KITTI[33] dataset through ORB-SLAM2 with the *Tarsier* device enabled as the accelerator. The KITTI dataset features a video sequence from two cameras mounted atop a car, as it drives through the German city of Karlsruhe. The two cameras allow a visual SLAM system to perform ***stereographic depth estimation***, where the distance to objects in the view can be estimated by calculating the positional discrepancy in the two views, just like human binocular vision.

The results are acceptable, but the system is not as stable as it is with ORB-SLAM2's custom, optimised feature detector/extractor. ORB-SLAM2 attempts to construct a map of the path that the camera follows by tracking how points move in the image as the vehicle moves. If the number of points that the detector is able to track from one frame to another drops too low, the SLAM system is unable to determine the transformation between the frames, and it loses its track. Certain regions of the KITTI dataset will cause ORB-SLAM2 to lose track of its position when using the *Tarsier* accelerator, while they are tracked correctly using the inbuilt detector/extractor. The *Tarsier* works adequately well on certain, easy regions of the dataset, but fails in these harder sequences.

Further work is required to improve the SLAM tracking robustness. The octree structure used in ORB-SLAM2's optimised detector achieves better coverage of the image by using dynamic

thresholding, allowing the detection threshold to vary across the image. This means that its detector will detect corners that the *Tarsier* fails to detect. In addition, the feature drops intrinsic to the *Tarsier* architecture can contribute to lower match rates, since the match will fail if only one of the pair of features is dropped.

The intimate knowledge of SLAM required to take a general feature extraction implementation and optimise it for tracking of robotic motion is, however, beyond the scope of this work.

A video of the operation of the ORB-SLAM2 system can be found at `https://youtu.be/VXK0lcJOHok`.

5.7 Conclusion

The design parameters allow the customisation of this hardware architecture to its environment. If it is desired to implement Harris–Stephens corner detection and ORB feature extraction on a resource-constrained FPGA, for reasons of cost or availability, aspects of the design, in particular the number of pyramid levels used and the precision of values in the Harris–Stephens feature detector, may be adjusted to minimise their requirements. On the other hand, if the highest performance is desired, and a suitable FPGA is available, the pyramid can be enlarged to 12 levels and more ORB modules added at each level. This may also improve SLAM tracking robustness.

The design is intended to be highly configurable, and the SystemVerilog parameters for the design are all exposed at the top level, for simple reconfiguration. The parameters selected for the *Tarsier* implementation on an Arria V GX have been chosen because they provide good performance, while still fitting into the hardware resources available.

The following chapter addresses generalisation of the techniques presented in this work to other problems, then discusses the achievements of this work, before closing with an investigation of further work that can be undertaken to improve the design.

Chapter 6

Generalisation, Conclusions and Future Work

This chapter considers approaches for generalisation of the image processing algorithms presented in this work, and specifically their implementation on an FPGA device. A conclusion section, examining the achievements of the work, follows, before the chapter closes with ideas for future work.

6.1 Generalisation of Results: Stencil Codes

Stencil codes are a family of algorithms employed for solving discrete problems in multiple dimensions. They can be used to solve problems as diverse as Conway's Game of Life, heat flow in three-dimensional objects and weather modelling.

Stencil codes are only applicable to problems on a uniform grid space, where the new value of each cell in the space is a function of the old values of the cells in the immediate neighbourhood. The same function is applied to every cell in the space, so the access patterns around all the cells are identical (with the possible exception of the boundary cells).

Stencil codes normally operate on spaces with one temporal dimension and either two or three spatial dimensions, although theoretically any number of spatial dimensions can be applied. The temporal dimension is ***causal***, meaning that functions are only allowed to depend on previous values, never on future values. This is a necessary condition, because the function itself is used to generate these future values.

FAST, Harris–Stephens, ORB, SIFT and other image processing algorithms are similar to stencil codes, in that they are applied to every window in an input domain, except that instead of updating the values for the stencil code states, they instead output a new stream of values, which is not written back but instead further processed.

Consider a k-dimensional array of elements, $\boldsymbol{S}_t$, of dimensions $(n_0, n_1, \ldots, n_{k-1})$, upon which our stencil code will operate, at a particular time t. I is the set of valid coordinates in the array. Each element in $\boldsymbol{S}_t$ can take values from a set S, representing the set of states.

$$I = \prod_{i=0}^{k-1} \{0, \ldots, n_i\} \tag{6.1}$$

$$\boldsymbol{p} \in I \tag{6.2}$$

$$t \in \mathbb{N}^0 \tag{6.3}$$

$$\boldsymbol{S}_t[\boldsymbol{p}] \in S \tag{6.4}$$

where $\mathbb{N}^0$ is the union of the set of natural numbers and zero, or $\{0, 1, 2, \ldots\}$.

The stencil region is defined by the set of points it operates on, and a function on the values

of these cells. The points are given by the set

$$s \in \prod_{i=0}^{l-1} \mathbb{Z}^k \tag{6.5}$$

that is, a set of l points, each of which is an offset from a central pixel at $\mathbf{0}$, and the function is given by

$$T : S^l \to S \tag{6.6}$$

This function T, termed the stencil function, takes the states of the l neighbouring cells and returns a new state for the central pixel.

The stencil function is applied to every element in $\boldsymbol{S}_t$, and calculates the element's new value, $\boldsymbol{S}_{t+1}[\boldsymbol{p}]$, from its neighbourhood.

The values in the cells of the space, represented by the set S, can themselves be discrete or continuous. Discrete values include examples like states in cellular automata, and continuous values include examples like pressure and air temperature in weather modelling.

6.1.1 Discrete-Domain Problems

Problems on a discrete domain, such as cellular automata and image processing, are the most straightforward to implement with stencil codes.

In the case of Conway's Game of Life, the space is an "infinite" 2-dimensional grid, the cells are each binary values, and the new value is a function of the neighbouring 8 cells. This function is applied to each 3×3 window in the space, and the resultant value is the new value of the centre cell of the window. Given the initial state of the automaton, $S_{t=0}$, applying the stencil code once yields the next state, $S_{t=1}$. Further applications of the stencil code generate later states.

Gaussian blur of images can also be implemented easily using a slight variation on a stencil code. The input image is the image to be blurred, and the function to apply across the input is a Gaussian blur filter response. In image processing, the time dimension does not represent a physical quantity, as it does in other problems, but instead represents the number of times

the filter is applied. Applying a Gaussian blur multiple times will have the effect of blurring the input image more. A stencil code implementation of a Gaussian blur would choose the number of iterations as a design parameter, and simply return the current system state after the desired number of iterations have been applied.

6.1.2 Continuous-Domain Problems

Continuous-domain partial differential equations, such as heat flow, can also be modelled with stencil codes by first discretising the domain, using a process like ***finite-element analysis***. Finite-element analysis involves chopping the domain up into small cubes and treating each of these as a discrete value. In a problem like heat flow, each cell in the stencil code, representing a cube of physical matter, transfers some of its value to nearby cells and receives some of their value on every update step, with the rate of change proportional to the difference in value. The value in this case represents heat.

The rules applied to the stencil code can vary across the domain. For instance, a part of the domain might represent an insulator, and thus its rate of heat flow will be very low. A different part might represent a good conductor, and there the rate of head flow will be fast.

6.1.3 Boundary Conditions

Stencil codes, by their nature, will only operate effectively on finite domains. Conway's Game of Life is traditionally evaluated over an infinite domain, and so some solution must be found to deal with the boundaries of the domain.

For some problems, like the Game of Life, it is acceptable to have a topologically toroidal world, where the left side is connected to the right side and the top to the bottom. Stencil codes operating along the margins of the domain are affected by cells on the far side, and cells can interact as if they are adjacent.

For modelling of physical problems, this approach is often inappropriate. Instead, boundary conditions can be constructed, where for instance the top edge of the domain can always have the same value. In the case of heat flow, this can represent a heat source.

6.1.4 FPGA Realisation

FPGAs are good candidates for realising many of these stencil codes, using a memory and a sliding window(see section 2.3.1), and have been used by other groups in the past[14]. The state is stored in the memory, and read out from the memory and into the sliding window. The stencil code is applied to the contents of the sliding window, and the results are written back into the memory. Each scan of the memory represents a single time step of the stencil code.

If the entire domain will fit into on-chip RAM, a single dual-ported RAM may be used for both reading and writing. The read port and write port point to different positions in the memory, and allow simultaneous read and write.

If the domain is too large to fit into on-chip RAM, it may be necessary to use off-chip SDRAM to store the state. Since SDRAM is single-ported, it is necessary to alternate reading and writing—both cannot be done simultaneously. A small cache of on-chip RAM will help to reduce latency, and allow more effective batching.

The sliding window is an ideal structure for accessing the regions of the image used by the stencil code, since it presents the entire neighbourhood at once as register values, not requiring random memory access. Additionally, the sliding window will always present the current state to the stencil code, not the new state. A naive stencil code applied to an array of cells in memory will update cells that are still to be used by future executions of the stencil code, corrupting the state.

The stencil code in question operates upon the input neighbourhood, returning the new value of the centre cell. This value is then written back to the memory, ready to be read out or to be updated yet again. The throughput of the stencil operation is critical to the speed of the FPGA design, since it limits the rate at which pixels can flow into the stencil code window. It is thus advisable to pipeline the stencil operation aggressively, since two executions of the stencil function are independent of one another.

Memory consumption of the sliding window can be large, and is the primary disadvantage of this architecture. An n-dimensional sliding window must store complete $(n-1)$-dimensional slices of the input domain in order to operate efficiently. For a one-dimensional domain, this means storing single points. For a two-dimensional domain, full rows of domain data must be stored, like in the Harris–Stephens detector. For a three-dimensional domain, entire planes of

input data must be stored. These slices must be stored in on-chip block RAM, and this can rapidly eat up all the available RAM on the device.

For example, a heat flow stencil code, evaluated to 16 bits of precision, with a window size of $3 \times 3 \times 3$, in a $1000 \times 1000 \times 1000$ domain, requires the storage of $3 \times 1000 \times 1000 \times 16$ bits, or 48 Mbits. The Arria V GX used in this work has only 19.8 MBits of on-chip block RAM, and so this particular problem is too large for it. The total SDRAM usage would also be 2 GB, which is much larger than the 256MB available on the Arria V GX starter kit board used in this work.

A smaller $200 \times 200 \times 200$ problem would fit comfortably into the available memory, however, at the cost of resolution.

A well-pipelined stencil code should be able to achieve a frequency of 250 MHz on an FPGA, allowing execution rates of around 250 million cells/s. Depending on the exact stencil function, this may be much faster than a CPU is able to manage, particularly if the window is very large, since this would require many RAM accesses on a CPU. For problems with very large windows, the memory locality enabled by the sliding window provides very fast access to the whole window content, where a CPU would have to access RAM.

6.1.5 Conclusion

The sliding window structure is an effective tool for applying stencil codes to multidimensional domain problems on an FPGA. It can yield very high speeds, at the cost of high memory usage. The speed advantage relative to a CPU is primarily dependent on the size of the window, however, with larger windows being more advantageous to the FPGA, thanks to the memory locality.

6.2 Conclusions

This work has presented a new hardware architecture for the acceleration of detection of Harris–Stephens corner features [3] and extraction of ORB feature descriptors [4]. While not the first such hardware architecture, this implementation makes more effective use of the hardware than the previous work [6], achieving much higher speeds and guaranteed worst-case performance, allowing the *Tarsier* device to serve in a hard real-time environment.

The usefulness of the ORB feature descriptor extractor has been improved significantly with the addition of multi-scale image processing, significantly increasing the quality of the feature matches achievable with the *Tarsier* device over other implementations [6]–[8].

The *Tarsier* accelerator has demonstrated high frame rates, allowing processing of 1920×1080 video at 60 fps (1080p60) in real time, processing each frame completely before the next arrives. The modifications to the standard algorithms are minor, despite this performance.

The design parameter space has been explored in detail, justifying the choices made in the design, as well as providing a guide to modifications that can be made in resource-constrained environments.

And finally, the core source code, the SystemVerilog that is compiled to construct the feature detector and extractor, has been released to the research community under a Mozilla Public Licence.

6.3 Future Work

6.3.1 Hardware matching

At the moment, one of the limiting factors to the usefulness of the Tarsier accelerator is that only the stages of feature detection and extraction are done in hardware—the process of matching, which is computationally expensive, is still done on the CPU as part of the ORB_SLAM2 framework [5].

As discussed in section 3.6, the Hamming distance operation required to match ORB descriptors lends itself well to an FPGA implementation, and a basic one was written for the work presented in [1]. The matching algorithm itself is straightforward; the complication is storage of descriptors from previous frames. It is impractical to stream the features from previous frames to the Tarsier device for matching, and so ideally, they would be stored on the device, for instance in the on-board DDR3 SDRAM.

A further complexity is the integration of the hardware feature matcher into ORB_SLAM2. The code as written operates directly on the features and performs its own specialised object recognition matching, and so significant changes would need to be made, or even a new SLAM system. Construction of a new SLAM system is beyond the scope of this work.

6.3.2 Further detection and scaling analysis

Several parameters of the feature detection and resampling subsystems have not yet been investigated in detail, namely, the size of the Gaussian window, the value used for κ, and the nature of the antialiasing filters used in the resampling process. These are likely to have a minimal effect, but should be investigated for completeness

6.3.3 Further drop analysis

In section 5.4, the module arrangement and buffering strategies were selected to maximise the number of features successfully processed at each level, and minimise the number dropped. What was not, however, analysed, was the impact that these dropped features have on the ability of the algorithm to reconstruct a scene properly. It appears that despite dropping some corners, the Tarsier works correctly as an accelerator for ORB-SLAM2, but further investigation is neces-

sary to establish the effect that dropping corners has on SLAM accuracy and tracking robustness.

6.3.4 Comparison with optimised GPU implementation

The GPU implementation tested for comparison with the *Tarsier* device was drawn directly from OpenCV without modification, and potentially did not exploit the GPU fully. Images were not streamed, but instead required the return of results before the next frame was input. A worthwhile target of further research would be optimising a GPU algorithm, such that it can deal with streamed images, and such that it fully utilises the hardware available.

6.3.5 Separate clock domains

The system presented in this work runs at the one clock frequency throughout. Certain regions of the system may, however, be able to run faster than this. For example, if the ORB modules are able to run at 200 MHz, while the rest of the system operates at 150 MHz, a separate clock domain could be created for the ORB modules. It would necessary to create clock-crossing logic to handle the mismatched clocks, particularly where the domains interface.

Having the ORB modules operate at a higher clock speed than the feature detector would be advantageous for performance, because the ORB module processing is decoupled from the pixel clock, and is not subject to the pixel stream clock enable line. This means that the ORB modules could process features in less than 256 pixel clock cycles, meaning that they can return to availability sooner.

6.3.6 Tiling

A major performance boost that has not been examined in this work is the application of ***tiling***. Tiling divides the input image into chunks, each of which is handled by a separate processing system. For instance, with two tiles, a separate image pyramid could be constructed for each half of the input image, and each of the levels in the pyramids would have an independent feature detection/extraction chain.

This has been implemented in computer vision before, in the Canny edge detector implemented in [35], but not for Harris–Stephens or for ORB.

Tiling at a factor of 2 can theoretically double the speed of a computer vision application,

because each tile only has to process approximately half as much input data, but this comes at the cost of increased logic usage, since all the hardware must be duplicated. The current FPGA being used in this work, an Arria V GX, does not have sufficient logic remaining to implement a second parallel system, and so a larger FPGA would need to be acquired for full testing.

One additional cost of tiling is in the boundary region. If the input image is simply halved, and each tile handles half the pixels as if they were a full image, an extra margin region will be created in the middle of the image. The ORB feature extractor has a margin of 18 pixels, and so this central stripe would be 36 pixels wide. No features could be detected in this region.

To mitigate this stripe, the regions covered by each tile need to overlap, such that they include pixels notionally on the other side of the dividing line. By widening the input fed to each module by 18 pixels, the central region is covered completely, while ensuring that the same feature will not be processed in both tiles.

An additional complication is ensuring that each tile is fed with pixels. This is relatively simple if the image is stored in on-board SDRAM, because the memory bandwidth is high enough to read from two locations at once.

If the image is being streamed in over PCI-express, as is the case for the Tarsier, it is more complicated, since the image data must be interleaved appropriately, so that the tiles take turns receiving image data.

The tiling factor can be further increased above 2, so long as sufficient logic is available. This means that there will be more overlapping regions, which lowers the performance boost available and increases the memory requirements, as shown in table 6.1. This table shows the tile widths and speedup for an image 1920 pixels wide. In narrower images, the performance boost will be smaller, since the overlap region will be a larger proportion of the image. The efficiency is also shown, which is defined as the speedup divided by the number of tiles.

Tiles	1	2	3	4	5	6
Tile width	1920	978	664	507	413	350
Speedup	1.00	1.96	2.89	3.79	4.65	5.49
Efficiency	1.00	0.98	0.96	0.95	0.93	0.91
Memory requirement	1.00	1.02	1.04	1.06	1.08	1.09

Table 6.1: Tile widths, speedups and efficiencies for a 1920-pixel wide image

Appendices

Appendix A

Overview of Hardware Description Language Modules

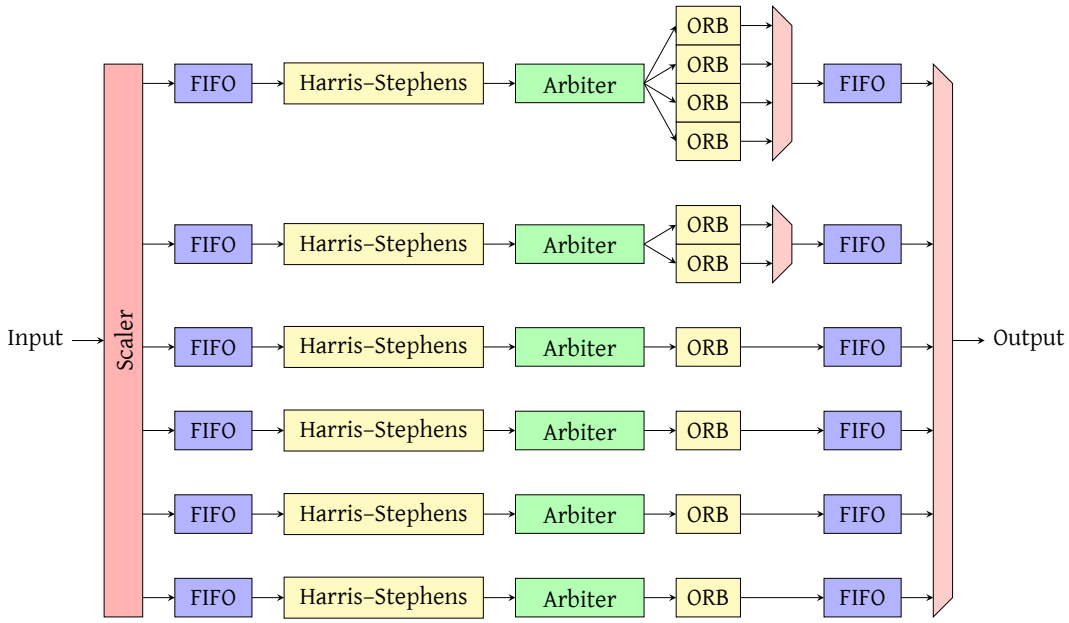


Figure A.1: The multilevel feature extraction system [2] ©2017 IEEE

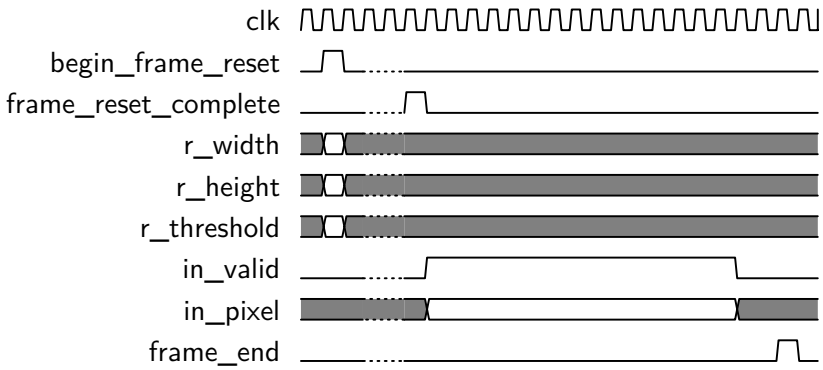


Figure A.2: Timing diagram from frame reset. The time between begin_frame_reset and frame_reset_complete

A.1 Concepts

Frame Reset

The frame reset signal is asserted at the beginning of the frame, and is responsible for coordinating changing the image dimensions of image processing components. The frame reset signal is asserted for all levels of the pyramid simultaneously, but with different dimensions. Two signals — `r_width` and `r_height` — accompany the frame reset, and are used to convey the appropriate width and height of that level. For certain modules, the frame reset additionally specifies the Harris–Stephens threshold to use for the incoming frame using `r_threshold`.

Each level has a second signal to signal the completion of the frame reset. Once the frame controller has received this completion signal from all the levels, the pixels begin to flow into the processing system.

The timing for this is shown in figure A.2. `begin_frame_reset` is asserted, as are `r_width`, `r_height` and `r_threshold`. Each of the modules that is dependent on the dimensions of the image is then expected to reconfigure itself accordingly, then assert the `frame_reset_complete` signal. Once all the modules in the system have asserted `frame_reset_complete`, the image data begins to flow through the system. As each module receives the last pixel in its image, it asserts the `frame_end` signal. Once all of these have been received by the central controller, the frame is deemed complete, and the next frame begins.

A.2 AdderTreePipelined

Source available on GitHub: *AdderTreePipelined.sv*

The `AdderTreePipelined` module represents a signed binary adder tree, summing a configurable-sized array of values, each of configurable bit-width, to yield a grand total. The summation is done as a binary tree, meaning that each adder adds two values together to yield an intermediate sum, and so the total number of adders required is $n - 1$, where n is the number of values in the input array. The addition is pipelined such that one addition is performed per pipeline stage, and the output is registered. A clock enable signal is also provided in order to pause the processing, to better synchronise with other modules. For an n-value input array, the total delay (in clock cycles) is $\lceil \log_2(n) \rceil$.

The module itself is written as a recursive generator, so each instantiation of the module represents only one addition, and its adder's inputs are calculated using generated submodules of half the input size. If a given instantiation of `AdderTreePipelined` has an input array of size 1, it simply delays its input for the appropriate number of clock cycles before forwarding it to the output. If the instantiation has input array size of more than 1, two recursive submodules are generated, whose input array sizes are half the size of the first instance, and whose outputs are summed together.

Parameters

`DATA_WIDTH`	The bit-width of each of the input values
`LENGTH`	The number of input values

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_addends`	input values (2's complement)
`in_advance`	clock enable
`out_sum`	output sum (2's complement)

A.3 BufferedCornersAndDescriptors

Source available on GitHub: *BufferedCornersAndDescriptors.sv*

`BufferedCornersAndDescriptors` serves as a wrapper around `CornersAndDescriptors`, providing FIFO queues to the input side of the module. This allows input to the underlying `CornersAndDescriptors` module to be delayed if the ORB modules are not ready for it.

As with `CornersAndDescriptors`, the input is a stream of pixels and the output is a stream of detected and extracted features.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`MAX_IMAGE_WIDTH`	the largest frame width this module can be configured to use (frame reset)
`MAX_IMAGE_HEIGHT`	the largest frame height this module can be configured to use (frame reset)
`COORD_BITS`	the bit-width of the coordinate signals
`MATRIX_BITS`	the number of bits to use for results in HarrisMatrix
`NONMAX_SCORE_BITS`	the number of bits to use for non-maximum suppression scores
`PARALLEL_MODULES`	the number of parallel ORB modules to instantiate
`BUFFERING_STRATEGY`	the buffering strategy to use. 0 means the "stall when all modules processing" strategy. 1 means the "stall when any module processing".
`FIFO_LEN`	the maximum length of the input FIFO

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_begin_frame_reset`	signals that the frame reset is beginning, and that `r_width` and `r_height` are valid
`r_width`	the image width of the incoming frame
`r_height`	the image height of the incoming frame
`r_threshold`	the Harris–Stephens corner detection threshold to use for the incoming frame
`in_valid`	signals that `in_pixel`, `in_x`, and `in_y` are valid
`in_pixel`	the incoming pixel
`in_x`	the x-coordinate of the incoming pixel
`in_y`	the y-coordinate of the incoming pixel
`in_consume`	signals that the current FIFO output is being dequeued
`in_mask`	allows particular ORB modules to be disabled. A set bit disables the corresponding module
`out_frame_reset_complete`	signals that the frame reset has been completed
`out_corner_count_increment`	signals that a corner has been detected, allowing an external counter to keep track of corner counts
`out_frame_end`	signals that all the pixels in the frame have passed through the module
`out_valid`	signals that `out_descriptor`, `out_feature_x`, and `out_feature_y` are valid
`out_descriptor`	the ORB descriptor for the output feature
`out_feature_x`	the x-coordinate of the output feature
`out_feature_y`	the y-coordinate of the output feature

A.4 CoordinateTransformer

Source available on GitHub: *CoordinateTransformer.sv*

The `CoordinateTransformer` module is used to calculate the effective full-scale position of a detected feature. Higher levels of the pyramid operate with smaller images than lower levels of the pyramid, and features coordinates output by them are in a different coordinate from from the original image. The `CoordinateTransformer` module is responsible for scaling these coordinates to the base scale and additionally performing a round-to-nearest, to preserve as much position information as possible.

The module is entirely combinational, and performs only shifts and adds (all the multiplications are by constants). See section 4.2.5 for details of the mathematics of unscaling.

Note that this module only supports a scaling ratio of $\frac{4}{5}$, and handles each dimension separately. Two modules are required for each pyramid level.

Parameters

`COORD_BITS`	the bit-width of the coordinate signals
`OCTAVE`	the octave of this level i.e. $\lfloor \frac{level}{3} \rfloor$
`SUBLEVEL`	the sublevel of this level i.e. $level \mod 3$

Input/Output

`in_coord`	the input coordinate (one dimension)
`out_coord`	the output coordinate

A.5 CornersAndDescriptors

Source available on GitHub: *CornersAndDescriptors.sv*

`CornersAndDescriptors` is the core of the feature detection and extraction system. Pixels enter at the input side and features exit at the output side. This module is instantiated once per level of the image pyramid, and contains the corner detector and one or more ORB modules.

`CornersAndDescriptors` counts pixels internally in order to generate the coordinates for its submodules. Since each submodule operates with a different input delay, each receives a different coordinate for a given input coordinate. This has the effect that the input coordinates are ignored, but they are useful for debugging.

`CornersAndDescriptors` is also the home of the main sliding window. This is a rectangular stripe of the image that contains all the raw pixel values needed for the various submodules. `HarrisCornersAndNonmax` takes a 3×3 window, which is used to calculate the image derivatives for Harris–Stephens corner detection. `ORBArbitrator` takes a 37-element column of pixels, which are inserted into the ringbuffers of each ORB module. The main sliding window is sized and positioned to fit both these input areas, with the effect that the "centre" pixel of the window, at virtual coordinate (0,0), is not actually present in the window when a corner is detected.

Parameters

LUMA_BITS	the bit-width of the pixel input
MAX_IMAGE_WIDTH	the largest frame width this module can be configured to use (frame reset)
MAX_IMAGE_HEIGHT	the largest frame height this module can be configured to use (frame reset)
COORD_BITS	the bit-width of the coordinate signals
MATRIX_BITS	the number of bits to use for results in HarrisMatrix
NONMAX_SCORE_BITS	the number of bits to use for non-maximum suppression scores
PARALLEL_MODULES	the number of parallel ORB modules to instantiate
BUFFERING_STRATEGY	the buffering strategy to use. 0 means the "stall when all modules processing" strategy. 1 means the "stall when any module processing".

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_begin_frame_reset`	signals that the frame reset is beginning, and that `r_width` and `r_height` are valid
`r_width`	the image width of the incoming frame
`r_height`	the image height of the incoming frame
`r_threshold`	the Harris–Stephens corner detection threshold to use for the incoming frame
`in_valid`	signals that `in_pixel`, `in_x`, and `in_y` are valid
`in_pixel`	the incoming pixel
`in_x`	the x-coordinate of the incoming pixel
`in_y`	the y-coordinate of the incoming pixel
`in_consume`	signals that the current FIFO output is being dequeued
`in_mask`	allows particular ORB modules to be disabled. A set bit disables the corresponding module
`out_frame_reset_complete`	signals that the frame reset has been completed
`out_corner_count_increment`	signals that a corner has been detected, allowing an external counter to keep track of corner counts
`out_frame_end`	signals that all the pixels in the frame have passed through the module
`out_request_stall`	signals that the module is applying backpressure to its input. The `BufferedCornersAndDescriptors` module will attempt to respect this signal, but if the FIFO is full, it will be ignored.
`out_valid`	signals that `out_descriptor`, `out_feature_x`, and `out_feature_y` are valid
`out_descriptor`	the ORB descriptor for the output feature
`out_feature_x`	the x-coordinate of the output feature
`out_feature_y`	the y-coordinate of the output feature

A.6 DimensionCalculator_4_5

Source available on GitHub: *DimensionCalculator_4_5.sv*

`DimensionCalculator_4_5` is used to calculate the dimensions of image pyramid levels at frame reset time. When a frame reset signal arrives, the `r_width` and `r_height` signals correspond to the width and height of the base-scale image, respectively. Each level of the pyramid above the base works with smaller images. This module calculates the size of those smaller images.

Given the dimension of an image pyramid level, this module will calculate the corresponding dimension for the level immediately above it, the next sublevel in that octave. These dimensions are in a ratio of approximately $5 : 4$, as detailed in section sec:scaling. The module actually performs a division, as opposed to multiplication by the reciprocal (1.25), since it is critical that the result be exact over the full range of input values.

The computation performed is

$$y = \frac{4(x-1)}{5} + 1$$

which compensates for the unevenness of the $\frac{4}{5}$ scaling filter. Division is performed by `SlowDividerUnsigned`, and takes many clock cycles (approximately 1 cycle for every bit of quotient). Since this is only performed once per frame, this is considered an acceptable delay.

Assert `in_valid` to commence calculation. `out_valid` will be asserted when the result is ready. If `in_valid` is asserted again, before computation is complete, the previous unfinished calculation will be cancelled.

Parameters

`COORD_BITS`	the bit-width of the coordinate signals

Input/Output

`clk`	clock signal
`in_valid`	signals that `in_dim` is valid, and that the calculation should commence
`in_dim`	the dimension to be scaled
`out_valid`	signals that the result in `out_dim` is ready
`out_dim`	the scaled dimension

A.7 FIFO

Source available on GitHub: *FIFO.sv*

The FIFO module represents a standard "first-in, first-out" queue, implemented with a ringbuffer. The length of the ring and the width of the data elements are customisable at compile time, and a signal is provided to indicate when the storage is full. The FIFO exhibits overwriting behaviour — if a word is written while the buffer is full, the oldest input is overwritten with the newest.

Parameters

`DATA_BITS`	the bit-width of each element
`ADDRESS_BITS`	the number of bits used for internal addressing. The maximum number of elements stored in the queue is equal to $2^{ADDRESS\_BITS} - 1$

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_write`	assert to write the value on `in_data` to the FIFO and advance the write head
`in_data`	the input to the FIFO
`in_read`	assert to advance the read head by one position
`out_data`	the output from the FIFO
`out_valid`	signals whether the value on `out_data` is valid. This will be set unless the FIFO is empty
`out_full`	signals whether the FIFO is full. If full, any further writes will cause data loss

A.8 HarrisCornersAndNonmax

Source available on GitHub: *HarrisCornersAndNonmax.sv*

The `HarrisCornersAndNonmax` module represents the combination of Harris–Stephens corner detection and 3×3 non-maximum suppression. The module takes a 3×3 window of image pixels as input and returns a corner determination for the centre point 4 rows and 23 columns later. This means that the corner score for pixel (x, y) arrives when the input window holds the contents of $(y + 3 \ldots y + 5, x + 22 \ldots x + 24)$. This assumes that the input window follows a raster scan, such as provided by the `SlidingWindow` module.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`MAX_IMAGE_WIDTH`	the largest frame width this module can be configured to use (frame reset)
`COORD_BITS`	the bit-width of the coordinate signals
`MATRIX_BITS`	the number of bits to use for results in HarrisMatrix
`NONMAX_SCORE_BITS`	the number of bits to use for non-maximum suppression scores

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`r_width`	the image width of the incoming frame
`r_threshold`	the Harris–Stephens corner detection threshold to use for the incoming frame
`in_valid`	signals that `in_window` is valid; serves as clock enable
`in_window`	a 3×3 window, used as input to the Harris detector
`out_is_corner`	signals whether a corner is detected, but delayed by 4 rows and 23 columns

A.9 HarrisCornersPipelined

Source available on GitHub: *HarrisCornersPipelined.sv*

The `HarrisCornersPipelined` module calculates Harris–Stephens corner scores for image pixels. When fed a raster scan of 3×3 windows of an image as input, it calculates the corner score and then outputs it with a delay of 2 rows and 17 columns.

The module feeds its input window into an instance of `HarrisMatrixPipelined`, which calculates the three distinct terms of the structure tensor. Each term goes into its own 7×7 sliding window. These three windows of matrix terms are blurred using a 7×7 binomial filter, implemented by multiplying each element by a constant (in effect a shift-and-add) and summing all 49 terms using an `AdderTreePipelined`. Since the sum of the coefficients of a binomial distribution is always a power of 2, the resultant sum can be shifted to the right to renormalise, instead of requiring a costly divider.

Finally, the Harris–Stephens score is calculated from the filtered structure tensor as described in section 2.2.1, using a k-value of 0.25 in order to yield simple arithmetic.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`MAX_IMAGE_WIDTH`	the largest frame width this module can be configured to use (frame reset)
`COORD_BITS`	the bit-width of the coordinate signals
`MATRIX_BITS`	the number of bits to use for results in HarrisMatrix

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`r_width`	the image width of the incoming frame
`in_valid`	signals that `in_window` is valid; serves as clock enable
`in_window`	a 3×3 window, used as input to the Harris detector
`out_score`	the corner score of the centre pixel, delayed 2 rows and 17 columns

A.10 HarrisMatrixPipelined

Source available on GitHub: *HarrisMatrixPipelined.sv*

The `HarrisMatrixPipelined` module calculates the Harris matrix (see section 2.2.1) of an image at a particular point.

The Harris matrix is given by

$$\begin{bmatrix} \left(\frac{\partial I}{\partial x}\right)^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \left(\frac{\partial I}{\partial y}\right)^2 \end{bmatrix}$$

using Sobel filters with response

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

for the x-derivative and y-derivative respectively.

The calculation is performed with a pipeline delay of 2 clocks, and calculates the results to a variable precision, configurable at compile time.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`MATRIX_BITS`	the number of bits to use for results in HarrisMatrix

Input/Output

`clk`	clock signal
`advance`	the clock enable signal, signals that window is valid
`window`	a 3×3 window of input pixels
`matrix`	a 2×2 matrix of the structure tensor, with precision given by `MATRIX_BITS`

A.11 MultilevelBarrier

Source available on GitHub: *MultilevelBarrier.sv*

The `MultilevelBarrier` module operates similarly to a software barrier, waiting for a number of subordinate tasks to complete before another task is able to continue. The module waits until every task has asserted its line of the `in_wait` signal before asserting `out_release`.

This module is used in the frame reset system to ensure that all levels of the pyramid have finished resetting and are ready for the incoming frame. The module is also robust to multiple wait signals on one line before any signal is received on another. A 4-bit counter is implemented for each line, with each assertion of the line incrementing the counter. Once all the counters are > 0, they are all decremented and `out_release` is asserted, allowing another task to proceed.

Parameters

`NUM_LEVELS`	the number of lines to implement

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_wait`	a bus with `NUM_LEVELS` signals, on which signals are sent to indicate completion of the subtasks
`out_release`	signals that all lines have been asserted, and that the main task may resume

Figure A.3: Caption

A.12 MultitapShiftRegister

Source available on GitHub: *MultitapShiftRegister.sv*

The `MultitapShiftRegister` module represents a shift register with multiple output taps along its length. The taps are evenly spaced, but their spacing is configurable at reset time.

The shift register is implemented using a wide RAM as the backing store, where the width of the RAM word is at least the data size multiplied by the number of taps.

As shown in figure A.3, on each update, one word is read from the RAM and then a word is written back to the same address. The new value is a left-shifted version of the old value, with the current input value in the low order bits, and the oldest data word discarded.

This allows the `MultitapShiftRegister` module to implement the backing storage for a sliding window. If the spacing between taps is set to the width of a row of image data, and the module is fed with pixels in a raster scan of the image, the values of the taps will represent a column of pixels from the input image.

If the shift register is configured to use a tap spacing less than the maximum, the memory write head will return back to address 0 before it reaches the end of the address space. For instance, if the maximum spacing is 1024 and the module is configured to use a spacing of 720, the head will update addresses in sequence $(\ldots 718, 719, 0, 1 \ldots)$. This allows the module to serve as the backing store for a sliding window on images of variable size.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

`DATA_BITS`	the bit-width of the input
`COORD_BITS`	the bit-width of the coordinate signals
`MAX_TAP_SPACING`	the maximum configurable distance between output taps (the actual distance used is the smallest power of two greater than or equal to this value)
`NUM_TAPS`	the number of output taps

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`r_tap_spacing`	the tap spacing that will be loaded on module reset
`in_valid`	signals that `in_data` is valid
`in_data`	the input value to be loaded into the shift register
`out_data`	an array of `NUM_TAPS` elements with the values of the shift register at each of the taps

A.13 NonmaxSuppression

Source available on GitHub: *NonmaxSuppression.sv*

The `NonmaxSuppression` module examines a window of values and determines whether the centre value is the largest value in the window. With a window size of 3×3, it is used to perform non-maximum suppression of Harris–Stephens corner scores, and ensure that only local maxima are returned from the corner detector. The centre value is also compared against a threshold, and will only be considered a local maximum if its value is higher than the threshold. This allows weak corners to be rejected.

The module is purely combinational, so there is no pipelining. Additionally, in order to break ties, if two equal values are adjacent, the value to the right or down is considered to be larger than the value to the left or up.

The centre pixel is defined to be that one at

$$\left(\left\lfloor \frac{WINDOW\_WIDTH}{2} \right\rfloor, \left\lfloor \frac{WINDOW\_HEIGHT}{2} \right\rfloor\right)$$

Parameters

`DATA_BITS`	the bit-width of the input
`WINDOW_WIDTH`	the width of the window
`WINDOW_HEIGHT`	the height of the window

Input/Output

`window`	the window to be examined
`threshold`	the minimum corner score required to accept a corner
`out`	signals whether the centre pixel is a local maximum

A.14 ORB2

Source available on GitHub: *ORB2.sv*

The ORB2 module is responsible for actually calculating ORB descriptors at detected feature locations. The module drives an ORBWindow module that stores the local neighbourhood of the feature and enables random access to offsets within that neighbourhood.

ORB2 contains a state machine that performs the various tasks of the ORB algorithm in order (see 3.4). First, the image moments calulated by ORBWindow are provided to SectorSel to determine the appropriate rotation angle. This rotation angle is used by the VecRotate modules to determine the appropriate rotated sample coordinate at which to sample the image. There are 256 pairs of samples that are accessed by the ORB2 module over the course of calculating the descriptor, and they are performed at a configurable rate, defaulting to one pair per clock cycle throughput. Once the samples have been acquired, they are compared and the result shifted into an output register.

The module may be configured to hold multiple ORBWindows, allowing it to perform multiple pairs of samples every clock, but typically the resources are better spent on extra ORB2 modules. With one window in a module, the latency is 266 clock cycles between commencement of calculation and availability of descriptor.

If columns of input pixels are received while the ORB2 module is calculating the descriptor for a feature, these cannot be written into the window, and thus the window will need to be refilled before it can be used again to calculate a descriptor.

Parameters

LUMA_BITS	the bit-width of the pixel input
BITS_PER_CLOCK	the number of ORBWindow modules to instantiate, and by extension, the number of descriptor bits that can be calculated per clock
COORD_BITS	the bit-width of the coordinate signals

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_valid`	signals that `in_col` and `in_go` are valid
`in_col`	a 37-element column of input pixels
`in_go`	signals the detection of a corner at the coordinate specified by (`in_x`,`in_y`) and triggers calculation of the descriptor. Additionally signals that `in_x` and `in_y` are valid
`in_x`	the x-coordinate of the detected feature
`in_y`	the y-coordinate of the detected feature
`out_valid`	signals that `out_descriptor`, `out_feature_x` and `out_feature_y` are valid
`out_descriptor`	256-bit value of the feature descriptor
`out_feature_x`	the x-coordinate of the output feature
`out_feature_y`	the y-coordinate of the output feature
`out_window_ready`	signals whether the module is currently available to calculate descriptors
`out_accepting_input`	signals whether the module is currently accepting pixel input. This will be set unless the module is currently processing.

A.15 ORBArbitrator

Source available on GitHub: *ORBArbitrator.sv*

The `ORBArbitrator` module is responsible for managing multiple ORB modules at a particular level of the image pyramid. It keeps track of the readiness of multiple modules, directs input pixel columns to those that are ready, and dispatches one module when a corner is detected. It contains a FIFO to buffer the output of the multiple ORB modules, and ensure that their output is not lost if later steps in the pipeline are not yet ready for the feature descriptors, while allowing the modules to return to service as soon as possible.

ORB modules can be in one of 3 broad states: filling, ready, and processing. A module in the filling state requires more pixel input before it enters the ready state and is able to generate descriptors. A module in the ready state will accept input to remain in the ready state, and is ready to process a detected feature. When it receives the signal to start processing, the module transitions into the processing state, at which point it is unable to accept input until the processing finishes. If input has arrived since it began processing, the module is out of sync, and must refill. If no input has arrived, however, the module is still in sync, and may transition directly back to ready.

The `ORBArbitrator` module manages the states of the modules. If a corner is received when at least one module is in the ready state, it will trigger the processing on that module. If no module is available, the corner must be dropped, and will not appear in the output.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`PARALLEL_MODULES`	the number of parallel ORB modules to instantiate
`BITS_PER_CLOCK`	the number of `ORBWindow` modules to instantiate, and by extension, the number of descriptor bits that can be calculated per clock
`COORD_BITS`	the bit-width of the coordinate signals
`BUFFERING_STRATEGY`	the buffering strategy to use. 0 means the "stall when all modules processing" strategy. 1 means the "stall when any module processing".

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_valid`	signals that `in_col` is valid
`in_col`	a 37-element column of input pixels
`in_is_corner`	signals the detection of a corner at the coordinate specified by (`in_x`,`in_y`) and may trigger calculation of the descriptor. Additionally signals that `in_x` and `in_y` are valid
`in_x`	the x-coordinate of the detected feature
`in_y`	the y-coordinate of the detected feature
`in_consume`	signals that the current FIFO output is being dequeued
`in_mask`	allows particular ORB modules to be disabled. A set bit disables the corresponding module
`out_valid`	signals that `out_descriptor`, `out_feature_x` and `out_feature_y` are valid
`out_descriptor`	the ORB descriptor for the output feature
`out_feature_x`	the x-coordinate of the output feature
`out_feature_y`	the y-coordinate of the output feature
`out_request_stall`	signals that the module is applying backpressure to its input

A.16 ORBMultiscale

Source available on GitHub: *ORBMultiscale.sv*

The `ORBMultiscale` module implements an image pyramid for the ORB feature detector/extractor system. All the scaling and coordinate transformation occurs within this module.

The input to the module is the input pixels and coordinates of the image base layer. These pixels are fed into the cascade of scalers (see section 4.2) to generate the inputs for each level of the pyramid. The inputs then flow into instances of `BufferedCornersAndDescriptors`, one for each level.

The scaler cascade is implemented using two instances of `Scale_4_5_Bilinear`, one for each sublevel of the first octave (excluding the base), and one instance of `Scale_1_2_Bilinear` for each other sublevel.

`ORBMultiscale` is additionally responsible for calculating the sizes of the downsamples levels of the pyramid. This calculation is performed using four instances of `DimensionCalculator_4_5`, one for each dimension of the two sublevels that have $\frac{4}{5}$ scaling ratios. (All levels above these are exactly half the height and width of another level, and this division can be done simply using right shift operations.) Since the calculation takes some time, and the result for the first instance (the dimensions of level 1) is required to begin the second instance (the dimensions of level 2), this is the primary component of the frame reset delay.

The module itself is implemented as a generate block. Each model is either the base level (level 0), or it derives its input from a scaled version of the input to a lower level. For each level, the source level's pixels are fed through the appropriate scaler, and then into an instance of `BufferedCornersAndDescriptors`. The relationships are shown in figure A.4.

When output features are received from the ORB modules, their coordinates are in the frame of the respective level of the pyramid. In order to be useful, all coordinates must be in the same, base scale, frame. This is done by passing the output coordinates through instances of `CoordinateTransformer`.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

LUMA_BITS	the bit-width of the pixel input
MAX_IMAGE_WIDTH	the largest frame width this module can be configured to use (frame reset)
MAX_IMAGE_HEIGHT	the largest frame height this module can be configured to use (frame reset)
COORD_BITS	the bit-width of the coordinate signals
NONMAX_SCORE_BITS	the number of bits to use for non-maximum suppression scores
MATRIX_BITS	the number of bits to use for results in HarrisMatrix
FIFO_LEN	the maximum length of the input FIFO

Level	Source Level	Relationship
0	Input	1
1	0	$\frac{4}{5}$
2	1	$\frac{4}{5}$
3	0	$\frac{1}{2}$
4	1	$\frac{1}{2}$
5	2	$\frac{1}{2}$
6	3	$\frac{1}{2}$
7	4	$\frac{1}{2}$
8	5	$\frac{1}{2}$

Figure A.4: The scaling relationships between levels

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_begin_frame_reset`	signals that the frame reset is beginning, and that `r_width` and `r_height` are valid
`out_frame_reset_complete`	signals that the frame reset has been completed
`r_width`	the image width of the incoming frame
`r_height`	the image height of the incoming frame
`r_threshold`	the Harris–Stephens corner detection threshold to use for the incoming frame
`in_valid`	signals that `in_pixel`, `in_x` and `in_y` are valid
`in_pixel`	the incoming pixel
`in_x`	the x-coordinate of the incoming pixel
`in_y`	the y-coordinate of the incoming pixel
`in_output_ready`	signals that the enclosing module wishes to consume the feature at the head of the output FIFO
`in_masks`	contains the enable masks for all the levels in the image pyramid
`out_corner_count_increments`	bus of signals, one for each level, to indicate the detection of a corner. Allows for an external counter to count the number of corners detected at each level
`out_frame_end`	signals that all the pixels in the frame have passed through the module
`out_valid`	signals that `out_descriptor`, `out_level`, `out_feature_x` and `out_feature_y` are valid
`out_descriptor`	the ORB descriptor for the output feature
`out_feature_x`	the x-coordinate of the output feature
`out_feature_y`	the y-coordinate of the output feature
`out_level`	the level of the detected feature

A.17 ORBWindow

Source available on GitHub: *ORBWindow.sv*

`ORBWindow` is the core pixel buffer for the `ORB` module. It stores a window of 37×37 pixels in a ring buffer and enables random access to their values. As described in section 3.4.2, incoming columns of pixels are written into one end of the active area of the ring buffer, and cause the head to advance one position. This module is also responsible for feeding the appropriate input to the `XMoment` and `YMoment` modules, which calculate the image moments for the rotation angle calculation.

The `ORBWindow` module can operate in read mode and in write mode. In read mode, input is ignored, and two two-dimensional asynchronous read ports are made available to the `ORB` module. In write mode, input is written into the ring buffer, and the read ports are inoperative. If input is received while the `ORBWindow` is in read mode, the window is marked as being 'dirty', or not up to date, and it must be flushed completely with new input columns before it can be put into read mode again. The current read operation can be completed first, however.

This module is completely agnostic to image dimensions, and so does not need to be reset in between frames.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`COORD_BITS`	the bit-width of the coordinate signals
`MOMENT_BITS`	the bit-width of the output image moments

Input/Output

`clk`	clock signal
`in_valid`	signals whether `in_col` is valid
`in_col`	a 37-element column of input pixels
`in_coord1`	read address for port 1 (range ($[-18, 18]$,$[-18, 18]$))
`in_coord2`	read address for port 2 (range ($[-18, 18]$,$[-18, 18]$))
`in_flush`	commence flush cycle manually (e.g. at start of frame)
`in_mode`	unset for read mode, set for write mode. Value must be held steady.
`out_patch_valid`	signals whether the contents of the patch are clean and ready to be read
`out_xmoment`	the current value of the x moment for the 37×37 window
`out_ymoment`	the current value of the y moment for the 37×37 window
`out_pix1`	the read value from port 1 (undefined if in write mode)
`out_pix2`	the read value from port 2 (undefined if in write mode)

A.18 RAMBlock

Source available on GitHub: *RAMBlock.sv*

`RAMBlock` represents a block of synchronous-write, asynchronous-read RAM. The dimensions are customisable at compile time, and the block has one port for write and one for read.

Parameters

`DATA_WIDTH`	the bit-width of a word of data
`ADDR_BITS`	the bit-width of the address lines. The total memory size is 2^{ADDR_BITS}.

Input/Output

`clk`	clock signal
`in_we`	write-enable for the write port
`out_read_data`	output from the read port
`in_read_addr`	address for the read port
`in_write_data`	input to the write port
`in_write_addr`	address for the write port

A.19 RecBinaryMux

Source available on GitHub: *RecBinaryMux.sv*

`RecBinaryMux` is a recursive pipelined structure, to allow multiple wide inputs to be multiplexed onto a single output, where ordering is unimportant. The structure is implemented as a binary tree, where each module has up to two inputs and a wide register. Each module can be either full or empty, and is aware of the states of its two children. An empty module, or one that will be taken from in this cycle, can take a value out of one of its full children, allowing a value to propagate from the input end of the tree to the output end. Because each module will take from one of its children by preference, it is possible for a value that entered the tree before another value to be output after it.

Parameters

`DATA_WIDTH`	the bit-width of a word of data
`LENGTH`	the total number of inputs to both of this module's subtrees

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`in_data`	a `LENGTH`-element array of input values for all this module's children
`in_valid`	a `LENGTH`-element array to signal validity for each of `in_data`
`in_ready`	signals to consume the input values from earlier modules
`out_data`	output to pass towards the end of the tree
`out_valid`	signals that `out_data` is valid
`out_ready`	input signal to consume the value in this module

A.20 Scale_1_2_Bilinear

Source available on GitHub: *Scale_1_2_Bilinear.sv*

Performs ratio-$\frac{1}{2}$ bilinear-filtered scaling on an input image. The input is a stream of pixels and coordinates, and the output is also a stream of pixels and coordinates, at approximately $\frac{1}{4}$ the average rate of the input.

As described in section 4.1.1, the output rate from the scaler modules is some fraction of the input rate. Whenever sufficient image information is available to calculate the value of the scaled output pixel, a pixel is emitted. The coordinates for the output pixel are generated internally by halving the input coordinates.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`MAX_IMAGE_WIDTH`	the largest frame width this module can be configured to use (frame reset)
`MAX_IMAGE_HEIGHT`	the largest frame height this module can be configured to use (frame reset)
`COORD_BITS`	the bit-width of the coordinate signals

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`r_width`	the image width of the incoming frame
`in_valid`	signals whether `in_pixel`, `in_x` and `in_y` are valid
`in_pixel`	the incoming pixel
`in_x`	the x-coordinate of the incoming pixel
`in_y`	the y-coordinate of the incoming pixel
`out_valid`	signals whether `out_pixel`, `out_x` and `out_y` are valid
`out_pixel`	the outgoing pixel
`out_x`	the x-coordinate of the outgoing pixel
`out_y`	the y-coordinate of the outgoing pixel

A.21 Scale_4_5_Bilinear

Source available on GitHub: *Scale_4_5_Bilinear.sv*

Performs ratio-$\frac{4}{5}$ bilinear-filtered scaling on an input image. The input is a stream of pixels and coordinates, and the output is also a stream of pixels and coordinates, at approximately $\frac{16}{25}$ the average rate of the input.

As described in section 4.1.1, the output rate from the scaler modules is some fraction of the input rate. Whenever sufficient image information is available to calculate the value of the scaled output pixel, a pixel is emitted. The coordinates for the output pixel are generated internally from the input coordinates.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

`LUMA_BITS`	the bit-width of the pixel input
`MAX_IMAGE_WIDTH`	the largest frame width this module can be configured to use (frame reset)
`MAX_IMAGE_HEIGHT`	the largest frame height this module can be configured to use (frame reset)
`COORD_BITS`	the bit-width of the coordinate signals

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`r_width`	the image width of the incoming frame
`in_valid`	signals whether `in_pixel`, `in_x` and `in_y` are valid
`in_pixel`	the incoming pixel
`in_x`	the x-coordinate of the incoming pixel
`in_y`	the y-coordinate of the incoming pixel
`out_valid`	signals whether `out_pixel`, `out_x` and `out_y` are valid
`out_pixel`	the outgoing pixel
`out_x`	the x-coordinate of the outgoing pixel
`out_y`	the y-coordinate of the outgoing pixel

A.22 SectorSel

Source available on GitHub: *SectorSel.sv*

The `SectorSel` module performs an approximate atan2 calculation, using 64 output angles (6 bits of precision), or approximately 5.6°steps. It is used in the ORB module to calculate the rotation angle from the image moments, to determine the orientation of a corner. As described in section 3.4.1, the calculation is done by multiplying the x-coordinate of a point by a list of precomputed factors, then comparing these to the y value.

The module takes 5 clock cycles to calculate the angle to 6 bits. In addition, the degenerate case where $x = y = 0$ is detected early, so that descriptor calculation can be aborted if the angle is undefined.

Parameters

`INPUT_BITS`	the bit-width of the input coordinates

Input/Output

`clk`	clock signal
`in_valid`	signals whether `in_x` and `in_y` are valid, additionally serves as clock enable
`in_x`	the x-coordinate of the point
`in_y`	the y-coordinate of the point
`out_sector`	the nearest angle (6 bits). `0` is defined to be upwards (negative y direction) and 16 points right (positive x direction)
`out_nan`	asserted if $in\_x = in\_y = 0$ and the angle is undefined

A.23 ShiftRegister

Source available on GitHub: *ShiftRegister.sv*

The `ShiftRegister` module is a configurable-length and -width single-tap shift register.

Parameters

`DATA_BITS`	the bit-width of the input
`LENGTH`	the length of the shift register

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`wr_en`	write- and clock-enable
`in`	input data
`out`	output data

A.24 SlidingWindow

Source available on GitHub: *SlidingWindow.sv*

The `SlidingWindow` module implements a 2D sliding window, as described in section 2.3.1. The window itself is implemented with an array of register-based shift registers, and the backing store is implemented with a `MultiTapShiftRegister`. The input to the module drives the bottom shift register row, while each of the other rows is driven by the output from one of the taps of the RAM-backed shift register.

The module makes the whole register-backed window available via its output, enabling fixed-offset access to the whole window.

The image width is reloaded upon synchronous reset, allowing this module to accept images of varying sizes without recompilation.

Parameters

`DATA_BITS`	the bit-width of the input
`WINDOW_NUM_ROWS`	the number of rows in the register window
`WINDOW_NUM_COLS`	the number of columns in the register window
`MAX_ROW_LENGTH`	the maximum runtime-configurable length of a row
`COORD_BITS`	the bit-width of the coordinate signals

Input/Output

`clk`	clock signal
`reset`	synchronous reset
`r_row_length`	the image width to configure on reset
`in_valid`	signals whether `in_data` is valid, also serves as clock enable
`in_data`	the input word
`out_window`	the register-backed window

A.25 SlowDividerUnsigned

Source available on GitHub: *SlowDividerUnsigned.sv*

`SlowDividerUnsigned` is an implementation of binary long division in hardware, generating one bit per clock cycle. The divider is used in the `DimensionCalculator_4_5` module, to ensure accuracy of calculated dimensions. Since the division only occurs twice per frame, it is considered acceptable to wait the delay, instead of implementing a faster, more complex algorithm, such as Newton-Raphson.

A dividend and divisor are input, and then, `DIVIDEND_BITS` cycles later, the quotient and remainder are output. If the divisor is `0`, the divide-by-zero bit is set.

Parameters

`DIVIDEND_BITS`	the bit-width of the dividend
`DIVISOR_BITS`	the bit-width of the divisor

Input/Output

`clk`	clock signal
`in_valid`	signals whether `in_dividend` and `in_divisor` are valid
`in_dividend`	the dividend
`in_divisor`	the divisor
`out_valid`	signals whether `out_error`, `out_quotient` and `out_remainder` are valid
`out_error`	set if a divide-by-zero occurred
`out_quotient`	the quotient
`out_remainder`	the remainder

A.26 UnsignedAdderTreePipelined

Source available on GitHub: *UnsignedAdderTreePipelined.sv*

Identical to `AdderTreePipelined`, just unsigned (see section A.2).

A.27 VectorRotate

Source available on GitHub: *VectorRotate.sv*

The `VectorRotate` module is used to rotate pairs of sample coordinates for indexing into an ORB window. The input is a vector representing an untransformed coordinate pair, as well as the sine and cosine values of the rotation angle from a lookup table, and the output is a rotated vector of the same magnitude as the input.

The input vector elements are 5-bit signed values ([-16,15]) and the sine and cosine values are 9-bit signed fixed-point values. In Q-notation, the input vector elements are in Q4.0 form, and the trigonometric values are in Q0.8.

Two intermediate values are calculated

$$a = x\cos\theta - y\sin\theta$$

$$b = x\sin\theta + y\cos\theta$$

which are signed 14-bit numbers with 8 fractional bits (Q5.8). These are then rounded to the nearest integer using Banker's rounding (break ties towards the nearest even number), and the signed 6-bit results are returned.

The latency of this module is 2 clock cycles, which is hidden by pipelining the pixel sample access in `ORB2`.

Parameters

none

Input/Output

`clk`	clock signal
`ix`	input x in signed Q4.0
`iy`	input y in signed Q4.0
`cos`	$\cos\theta$ in signed Q0.8
`sin`	$\sin\theta$ in signed Q0.8
`ox`	output x in signed Q5.0
`oy`	output x in signed Q5.0

Bibliography

[1] J. Weberruss, L. Kleeman and T. Drummond, 'ORB Feature Extraction and Matching in Hardware', in *Proceedings of the Australiasian Conference on Robotics and Automation*, Australian Robotics and Automation Association, 2015 (cit. on pp. 4, 13, 15, 86, 157).

[2] J. Weberruss, L. Kleeman, D. Boland and T. Drummond, 'FPGA acceleration of multilevel ORB feature extraction for computer vision', in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056856 (cit. on pp. 4, 15, 26, 63, 92, 101, 121, 147, 161).

[3] C. Harris and M. Stephens, 'A combined corner and edge detector', in *Alvey vision conference*, Manchester, UK, vol. 15, 1988, p. 50 (cit. on pp. 12, 19, 20, 156).

[4] E. Rublee, V. Rabaud, K. Konolige and G. Bradski, 'ORB: an efficient alternative to SIFT or SURF', in *IEEE International Conference on Computer Vision (ICCV), 2011*, IEEE, 2011, pp. 2564–2571 (cit. on pp. 13, 54, 55, 57, 59, 72, 82, 86, 91, 156).

[5] R. Mur-Artal and J. D. Tardós, 'ORB-SLAM2: An open-source SLAM system for monocular, stereo, and RGB-D cameras', *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017 (cit. on pp. 13, 144, 157).

[6] K. Lee, 'A design of an optimized ORB accelerator for real-time feature detection', *International Journal of Control & Automation*, vol. 7, no. 3, 2014 (cit. on pp. 15, 61, 89, 156).

[7] R. de Lima, J. Martinez-Carranza, A. Morales-Reyes and R. Cumplido, 'Improving the construction of ORB through FPGA-based acceleration', *Machine Vision and Applications*, vol. 28, no. 5-6, pp. 525–537, 2017 (cit. on pp. 15, 84, 85, 89, 156).

[8] R. Sun, P. Liu, J. Wang, C. Accetti and A. A. Naqvi, 'A 42fps full-HD ORB feature extraction accelerator with reduced memory overhead', in *International Conference on Field Programmable Technology (ICFPT), 2017*, IEEE, 2017, pp. 183–190 (cit. on pp. 15, 84, 89, 156).

[9] J. Canny, 'A computational approach to edge detection', in *Readings in Computer Vision*, Elsevier, 1987, pp. 184–203 (cit. on p. 18).

[10] E. Rosten and T. Drummond, 'Fusing points and lines for high performance tracking', in *Tenth IEEE International Conference on Computer Vision (ICCV), 2005*, IEEE, vol. 2, 2005, pp. 1508–1515 (cit. on pp. 19, 24).

[11] H. P. Moravec, 'Obstacle avoidance and navigation in the real world by a seeing robot rover.', Stanford University, Tech. Rep., 1980 (cit. on p. 21).

[12] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama and P. Manneback, 'A Multi-Resolution FPGA-Based Architecture for Real-Time Edge and Corner Detection', *IEEE Transactions on Computers*, vol. 63, no. 10, pp. 2376–2388, Oct. 2014, ISSN: 0018-9340. DOI: 10.1109/TC.2013.130 (cit. on pp. 27, 36, 40, 43, 46, 48, 49).

[13] P.-Y. Hsiao, C.-L. Lu and L.-C. Fu, 'Multilayered image processing for multiscale Harris corner detection in digital realization', *IEEE Transactions on Industrial Electronics*, vol. 57, no. 5, pp. 1799–1805, 2010 (cit. on pp. 27, 36, 37, 40, 48, 49).

[14] H. Yu and M. Leeser, 'Automatic sliding window operation optimization for FPGA-based computing boards', in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2006. FCCM'06.*, IEEE, 2006, pp. 76–88 (cit. on pp. 27, 154).

[15] A. Amaricai, C.-E. Gavriliu and O. Boncalo, 'An FPGA sliding window-based architecture Harris corner detector', in *24th International Conference on Field Programmable Logic and Applications (FPL), 2014*, IEEE, 2014, pp. 1–4 (cit. on pp. 37, 40, 46, 48, 49).

[16] I. Sobel and G. Feldman, 'A 3x3 isotropic gradient operator for image processing', *a talk at the Stanford Artificial Project*, pp. 271–272, 1968 (cit. on p. 41).

[17] D. G. Lowe, 'Object recognition from local scale-invariant features', in *The proceedings of the seventh IEEE international conference on Computer vision (ICCV), 1999.*, IEEE, vol. 2, 1999, pp. 1150–1157 (cit. on p. 53).

[18] M. Calonder, V. Lepetit, C. Strecha and P. Fua, 'BRIEF: Binary robust independent elementary features', in *European conference on computer vision*, Springer, 2010, pp. 778–792 (cit. on p. 54).

[19] OpenCV authors. (Feb. 2018). Utility and system functions and macros, [Online]. Available: `https://docs.opencv.org/3.4.1/db/de0/group__core__utils.html` (visited on 06/03/2018) (cit. on pp. 55, 72).

[20] V. Bonato, E. Marques and G. A. Constantinides, 'A parallel hardware architecture for scale and rotation invariant feature detection', *IEEE transactions on circuits and systems for video technology*, vol. 18, no. 12, pp. 1703–1712, 2008 (cit. on p. 60).

[21] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao and W. Feng, 'An architecture of optimised SIFT feature detection for an FPGA implementation of an image matcher', in *International Conference on Field-Programmable Technology, 2009. FPT 2009.*, IEEE, 2009, pp. 30–37 (cit. on p. 60).

[22] R. de Lima, J. Martinez-Carranza, A. Morales-Reyes and R. Cumplido, 'Accelerating the construction of BRIEF descriptors using an FPGA-based architecture', in *International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2015*, IEEE, 2015, pp. 1–6 (cit. on pp. 60, 84).

[23] T. Hu and T. Ikenaga, 'FPGA implementation of high frame rate and ultra-low delay vision system with local and global parallel based matching', in *Fifteenth IAPR International Conference on Machine Vision Applications (MVA), 2017*, IEEE, 2017, pp. 286–289 (cit. on p. 61).

[24] P. Viola and M. Jones, 'Rapid object detection using a boosted cascade of simple features', in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001. CVPR 2001.*, IEEE, vol. 1, 2001, pp. I–I (cit. on p. 66).

[25] (). Intel haswell, [Online]. Available: `https://www.7-cpu.com/cpu/Haswell.html` (visited on 25/03/2018) (cit. on p. 81).

[26] Intel Corporation. (). Arria V GX Starter Kit, [Online]. Available: `https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-arria-v-starter.html` (visited on 25/03/2018) (cit. on p. 81).

[27] J. E. Volder, 'The CORDIC trigonometric computing technique', *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959 (cit. on p. 85).

[28] S. Tanimoto and T. Pavlidis, 'A hierarchical data structure for picture processing', *Computer Graphics and Image Processing*, vol. 4, no. 2, pp. 104–119, 1975 (cit. on p. 91).

[29] Microsoft Corporation. (). Texture filtering with mipmaps, [Online]. Available: `https://msdn.microsoft.com/en-us/library/windows/desktop/bb206251(v=vs.85).aspx` (visited on 22/03/2018) (cit. on p. 91).

[30] C. E. Shannon, 'Communication in the presence of noise', *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949 (cit. on pp. 93, 96).

[31] R. Keys, 'Cubic convolution interpolation for digital image processing', *IEEE transactions on acoustics, speech, and signal processing*, vol. 29, no. 6, pp. 1153–1160, 1981 (cit. on p. 95).

[32] C. E. Duchon, 'Lanczos filtering in one and two dimensions', *Journal of applied meteorology*, vol. 18, no. 8, pp. 1016–1022, 1979 (cit. on pp. 95, 96).

[33] A. Geiger, P. Lenz, C. Stiller and R. Urtasun, 'Vision meets Robotics: The KITTI Dataset', *International Journal of Robotics Research (IJRR)*, 2013 (cit. on pp. 110, 147).

[34] Oxford Visual Geometry Group. (2004). Affine covariant regions datasets, [Online]. Available: `http://www.robots.ox.ac.uk/~vgg/data/data-aff.html` (visited on 17/01/2017) (cit. on p. 110).

[35] Q. Xu, C. Chakrabarti and L. J. Karam, 'A distributed Canny edge detector and its implementation on FPGA', in *Digital Signal Processing Workshop and IEEE Signal Processing Education Workshop (DSP/SPE), 2011 IEEE*, IEEE, 2011, pp. 500–505 (cit. on p. 158).