University of Strathclyde Glasgow

On the Efficient Implementation of the Line Hough Transform for Embedded Vision Systems

David Northcote

A thesis submitted for the degree of Doctor of Philosophy

Department of Electronic and Electrical Engineering

University of Strathclyde, Glasgow

July 28, 2023

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.50. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

Signed: David Northcote

Date: July 28, 2023

Abstract

Many embedded applications increasingly employ computer vision systems to perform visual inspection tasks. Vision systems allow a computer to see and interpret its environment to automate procedures using image processing algorithms. An emerging challenge for modern vision systems is to meet the processing demands of larger image resolutions and higher video frame rates. Field Programmable Gate Arrays (FPGAs) are commonly selected to accelerate embedded vision algorithms as they are widely available, cost-effective, reprogrammable, and energy efficient. They also offer parallel processing capabilities that accelerate algorithms and tasks to achieve low latency processing and high data throughput.

Line detection in digital images is essential for many embedded applications, such as lane detection in Advanced Driver Assistance Systems (ADAS) or powerline and railroad inspection using Unmanned Aerial Vehicles (UAVs). The Line Hough Transform (LHT) is a well-known technique for accurately detecting lines in digital images. The LHT achieves line detection by mapping edge pixels to a discrete array known as the Hough Parameter Space (HPS). Peaks form in the HPS that correspond to lines in the original image. Although the LHT is very robust to noise and can detect partially occluded lines, it is also highly computational and demands significant memory resources to store the HPS. Previous studies describe software optimisations to reduce the complexity of the LHT and present FPGA architectures that accelerate its computation. In many FPGA implementations, the memory consumption of the HPS is significant, making the LHT unsuitable for memory-constrained FPGA designs. Large image resolutions, such as 1920×1080 pixels, exacerbate this issue, demanding considerable memory resources to store the HPS.

Abstract

This thesis examines the memory consumption of the LHT algorithm in FPGAs and presents a novel research platform for LHT architectures named the Hough Evaluation Platform (HEP). The HEP can be used to design custom LHT architectures and validate them on the physical target device. This powerful validation technique improves the testing of hardware architectures beyond software simulations. The HEP can also accurately calculate the time taken for an LHT architecture to process a digital image. Furthermore, this thesis presents two novel techniques to reduce the memory requirements of the LHT, which are named the Symmetric LHT and the Angular Regions LHT (ARLHT). The FPGA architectures presented for each technique target Full High Definition (FHD) video, i.e. a resolution of 1920×1080 pixels at 60 frames per second (fps). The Symmetric LHT uses architectural optimisations to reduce the memory consumption of the HPS in FPGA devices. The proposed design employs memory bit-packing schemes to reduce on-chip memory resources by approximately 33% without affecting the accuracy of line detection. The ARLHT decreases the memory consumption of the LHT by using a lossy compression scheme to store the HPS. This scheme compresses the HPS by dividing the input image into angular regions before applying the LHT. The ARLHT architecture requires approximately 53% fewer memory resources than the standard LHT.

Acknowledgements

Firstly, I would like to thank my supervisors, Dr. Louise Crockett and Dr. Paul Murray, for giving me the opportunity to pursue a PhD and for their unwavering support in achieving my research goals and career aspirations. I am grateful to them for sparking my interest in FPGAs and image processing and for their mentorship, advice, and technical assistance throughout my doctoral studies. I want to express my gratitude to both of you for your patience and kindness and for generously sharing your time and experience with me. I also want to express my appreciation to Prof. Bob Stewart for his invaluable guidance and insights on this work and its research publications.

I would like to extend my gratitude to my friends and colleagues in the StrathSDR group at the University. You are all the most talented and dedicated people I have had the pleasure of working with, and it has been a privilege to work alongside you all over the years. I would like to give special thanks to Douglas Allan and Shawn Kalade for their friendship and encouragement during my studies. Each of you provided me with essential technical support and assistance, for which I am immensely grateful. I would also like to thank my family and friends for their unwavering support during this undertaking. Your kindness and encouragement have motivated me to persevere and complete this work. Finally, I would like to extend a special thank you to my partner, Kirsty McNair. Thank you for your unconditional love, support, and patience and for pushing me to see this through to the end.

This work was supported by Xilinx, Inc (now AMD) and the Engineering and Physical Sciences Research Council (EPSRC) grant EP/M508159/1.

Contents

Declaration i

Abstract ii

Acknowledgements iv

List of Figures x

List of Tables xxii

List of Acronyms xxvi

1 Introduction 1

1.1 Research Background 1

1.2 Research Aims and Objectives 3

1.3 Original Contributions 4

1.4 Publications and Awards 6

1.5 Thesis Structure 7

2 Embedded Vision Systems 9

2.1 Introduction 9

2.2 Embedded Hardware Platforms 10

2.3 Overview of FPGA Technology 11

2.3.1 The Logic Fabric 12

2.3.2 The DSP48E2 Slice 14

2.3.3 Block RAM Tiles 15

2.3.4 Architecture Evaluation 17
2.4 Overview of the Zynq MPSoC 18
2.4.1 The Zynq MPSoC Architecture 19
2.4.2 The Advanced eXtensible Interface 20
2.4.3 The PS and PL Interface 21
2.4.4 Data Movement and Communication 22
2.5 Design Tools and Development Board 23
2.5.1 MathWorks *HDL Coder* 24
2.5.2 Intellectual Property (IP) Cores 25
2.5.3 The Vivado Design Suite 26
2.5.4 PYNQ: The Python Productivity for Zynq Framework 27
2.5.5 The ZCU104 Development Board 31
2.6 FPGA & SoC Vision Architectures 32
2.6.1 Hardware Accelerated Image Processing 32
2.6.2 Stream Processing 35
2.6.3 Pixel Intensity 35
2.6.4 Image Representation 36
2.6.5 Local Filters 37
2.6.6 Edge Detection 41
2.6.7 Gradient Orientation 45
2.6.8 Binary Morphology 46
2.7 Summary 50

3 The Hough Transform 52
3.1 Introduction 52
3.2 Line Detection 54
3.2.1 The Glide Reflection 56
3.2.2 Memory Requirements 56
3.2.3 Vote Accumulation 58
3.2.4 Line Reconstruction 62
3.3 Literature Review 63

3.3.1 Software Implementations . . . 63
3.3.2 FPGA Architectures and Development Tools . . . 71
3.3.3 Discussion . . . 77
3.3.4 Key Findings . . . 78
3.4 Embedded Applications . . . 79
3.4.1 Advanced Driver Assistance Systems . . . 80
3.4.2 Unmanned Aerial Vehicles . . . 82
3.4.3 Industrial Inspection . . . 84
3.4.4 Wireless Communications . . . 85
3.4.5 Discussion of Requirements . . . 87
3.4.6 Key Findings . . . 89
3.5 Conclusion . . . 91

4 The Hough Evaluation Platform **92**
4.1 Introduction . . . 92
4.2 Evaluation Platform Design . . . 93
4.2.1 The Hough Inspection Unit . . . 95
4.2.2 The Hough Performance Analyser . . . 97
4.2.3 Visualisation and Analysis . . . 99
4.3 Embedded System Integration . . . 101
4.3.1 The HDL Coder Reference Design . . . 102
4.3.2 The IP Integrator Block Design . . . 108
4.3.3 Control and Analysis Software . . . 109
4.3.4 HEP System Integration Workflow . . . 112
4.4 Analysis and Evaluation . . . 117
4.4.1 Implementation Results . . . 117
4.4.2 Processing Time Results . . . 119
4.4.3 Architecture Validation and Testing . . . 119
4.4.4 Comparison with FPGA-in-the-Loop Simulation . . . 125
4.4.5 Discussion of Results . . . 126
4.5 Conclusion . . . 127

5 A Symmetric Hough Kernel and Bit-Packed Accumulator 130
5.1 Introduction . . . 130
5.2 The Symmetric LHT . . . 131
5.2.1 Resource Sharing . . . 132
5.2.2 Spatial Domain Symmetry . . . 133
5.2.3 Parallel Pixel Processing . . . 136
5.2.4 The Bit-Packed Accumulator . . . 137
5.2.5 The Voting Scheme . . . 141
5.3 Symmetric LHT Architecture Design . . . 141
5.3.1 The Pixel Packing System . . . 142
5.3.2 Coordinate Calculator . . . 144
5.3.3 Phase Controller . . . 145
5.3.4 Symmetric Hough Kernel . . . 146
5.3.5 Accumulator Controller . . . 148
5.3.6 Accumulator Circuit Design . . . 152
5.4 Analysis and Evaluation . . . 156
5.4.1 Implementation Results . . . 156
5.4.2 Processing Time Results . . . 158
5.4.3 Architecture Validation and Testing . . . 159
5.4.4 Comparison with Previously Published Work . . . 161
5.4.5 Discussion of Results . . . 163
5.5 Conclusion . . . 165

6 The Angular-Regions Line Hough Transform 167
6.1 Introduction . . . 167
6.2 The ARLHT Algorithm . . . 168
6.2.1 The Memory-Compressed HPS . . . 169
6.2.2 The Voting Scheme . . . 170
6.2.3 Peak Separation and Detection . . . 171
6.2.4 Line Reconstruction . . . 172
6.2.5 Memory Requirements . . . 173

6.3 The ARLHT Architecture Design 174
6.3.1 Preprocessing and Segmentation 175
6.3.2 Hough Kernel and Adjusted Orientation 175
6.3.3 Accumulator Circuit Design 177
6.3.4 RBM Circuit Design 182
6.3.5 Peak Separation and Detection Circuit Design 185
6.4 Analysis and Evaluation 186
6.4.1 Implementation Results 186
6.4.2 Processing Time Results 187
6.4.3 Architecture Validation and Testing 188
6.4.4 Comparison with Related Works 190
6.4.5 Limitations and Discussion of Results 192
6.5 Conclusion 193

7 Conclusions 196

7.1 Resume 196
7.2 Discussion of Results 198
7.3 Key Conclusions 201
7.4 Future Work 202

A CORDIC 204

A.1 The Circular CORDIC Algorithm 205
A.2 The Scaling Factor 207
A.3 The Circular CORDIC Equations 208
A.4 Region of Convergence 209
A.5 Vectoring Mode 211
A.6 Example: Circular CORDIC in Vectoring Mode 213
A.7 Summary 214

B Parallel LHT Implementation 215

B.1 MathWorks *HDL Coder* Blocks 215
B.2 Architecture Overview 218

B.3 Parallel LHT Kernel 219
B.4 Parallel LHT Accumulator 222
B.5 Standard LHT Software Model 227
B.6 Summary 229

C Symmetric LHT Validation Results 230

D ARLHT Validation Results 233

References 236

List of Figures

2.1 An overview of the AMD Kintex UltraScale+ FPGA architecture. . . . 12
2.2 An illustration of a CLB and switch matrix. The logic elements contained inside a slice are also shown. 13
2.3 Simplified architecture of the DSP48E2 slice. 14
2.4 36 Kb BRAM with dual-port interface (left), two 18 Kb BRAMs formed by separating a 36 Kb BRAM in half (right). 15
2.5 Simplified architecture of the RAMB36E2 primitive configured to operate in TDP mode (left), and SDP mode (right). Some ports have been removed or combined to simplify the diagram. 16
2.6 An overview of the Zynq MPSoC device architecture. 19
2.7 An example of a vision system with an HDMI interface and hardware accelerator implemented on the Zynq MPSoC device. The hardware accelerator can communicate with the external memory system and use it to buffer data and store run-time information. 22
2.8 A basic workflow diagram illustrating the relationship between the Zynq MPSoC development tools used in this work. 23
2.9 A subset of the MathWorks *HDL Coder* blockset (left). The product block and associated parameters are shown (right). 24
2.10 An AXI DMA IP core and its associated configuration properties. This image was generated from Vivado's IP catalogue. 26

2.11 An example of interconnecting two IP cores using the IP Integrator tool. The communication interface shown in the diagram uses the AXI4-Stream protocol. 27
2.12 The layers in the software stack of the PYNQ framework. 28
2.13 An example of a Jupyter Notebook containing rich markdown text cells, executable Python code cells, and output plots for visualisation. 29
2.14 An example of a radio introspection application for RFSoC PYNQ. . . . 30
2.15 The ZCU104 development board that will be used throughout this thesis as the target platform for implementing LHT architecture designs. . . . 31
2.16 An embedded vision system implemented on an FPGA. 33
2.17 An embedded vision system implemented on the Zynq MPSoC device. . 34
2.18 Scanning an $M \times N$ image (left), data stream of the original image (right). 35
2.19 A greyscale image of a radio tuner (a) that can be smoothed (b) or sharpened (c) using a local filter. Further information on local filter operations that can smooth or sharpen digital images can be found in [66]. 37
2.20 A 3×3 spatial filter window $w(s, t)$ centred on a pixel in an image $f(x, y)$. 38
2.21 An architecture for implementing a spatial filter on an FPGA. This technique arranges row buffers in parallel with the filter window. 40
2.22 An architecture for nearest neighbour extrapolation border management in FPGA implementations of local filters. 41
2.23 The H directional gradient (a) and V directional gradient (b) are used in (2.7) to obtain the gradient magnitude (c) of the radio tuner image. The corresponding edge image (d) is obtained by thresholding the gradient magnitude. 43
2.24 An FPGA architecture that applies the Sobel operators to a greyscale image. The architecture design uses separable filters and an efficient gradient magnitude calculation to minimise resource consumption [67]. . 44
2.25 FPGA architecture of unrolled CORDIC [74]. 45
2.26 An example of a binary set X (left) and structuring element B (right). . 46

2.27 The examples above present the erosion (a) and dilation (b) of the binary set X by the structuring element B. The dashed red outline shows the borders of the original object pixels to help visualise the effects of erosion and dilation on the input binary set. 48
2.28 An FPGA architecture that applies dilation or erosion to an image using a 3×3 structuring element. 48
2.29 The examples above present the opening (a) and closing (b) of the binary set X by the structuring element B. The dashed red outline shows the borders of the original object pixels to help visualise the effects of applying the operations on the input binary set. 49

3.1 The general HT process for extracting an object in a binary image. . . . 53
3.2 The slope-intercept representation of a straight line is used to map points, (x_i, y_i) and (x_j, y_j), in the spatial domain (left) to the ab-plane (right). The points are mapped using (3.1) over a set of real values, a. . 54
3.3 The normal representation of a straight line is used to map points, (x_i, y_i) and (x_j, y_j), in the spatial domain (left) to the $\rho\theta$-plane (right). The points are mapped using (3.2) over a set of real values, θ. 55
3.4 Points (x_i, y_i) and (x_j, y_j) in the spatial domain (left) are mapped to the $\rho\theta$-plane (right) using (3.2). The size of the $\rho\theta$-plane is halved using the glide reflection given in (3.3). 56
3.5 The spatial image domain of $M \times N$ pixels, containing one edge pixel at the bottom right corner, which has been enlarged for visualisation purposes (left). The corresponding HPS containing votes that have been mapped using (3.2), presented in top-down view (right). The image origin is located at the top left corner of the image. 57
3.6 The spatial image domain of 100×100 pixels, containing five edge pixels, which have been enlarged for visualisation purposes (left). The corresponding HPS containing votes that have been mapped using (3.2), presented in top-down view (right). 59

3.7 A colour image of a chess board (a) that has been converted to a greyscale image (b). Sobel operators are applied to the greyscale image to produce the gradient magnitude image (c) and thresholding is applied to produce an edge image (d). 60

3.8 The HPS for the chessboard edge image in Figure 3.7d. The HPS is shown from the top-down (a) and isometric view (b). 61

3.9 The HPS for the chessboard edge image in Figure 3.7d. The HPS is shown from the θ-axis (a), and ρ-axis (b). 62

3.10 Infinite line reconstruction in (a), finite line reconstruction in (b), overlay of the reconstructed image and colour image in (c), and overlay of the reconstructed image and greyscale image in (d). 64

3.11 The gradient orientation image of the chessboard (left), and a magnified region (right). The intensity of each pixel corresponds to its gradient orientation value. 65

3.12 Four HPS results for the chessboard edge image in Figure 3.7d, using the Gradient LHT. The HPS when $\lambda = 0$ (a), $\lambda = 44$ (b), $\lambda = 88$ (c), and $\lambda = 134$ (d). 66

3.13 The sub-images of the chessboard edge image in (a), are used to vote in the memory-compressed HPS in (b). 69

3.14 Four RBM memory arrays for each sub-image of the chessboard edge image. 70

3.15 Overview of an FPGA architecture of the LHT that primarily uses CORDIC [85]. 72

3.16 An illustration of the Hybrid-Log Hough kernel for computing the Hough parameters. 73

3.17 An unrolled LHT architecture that uses shift and add operations to compute the Hough parameters. The variables ρ_a and ρ_b are used to denote signal paths in the architecture design. See the corresponding publication in [13] for more information. 73

3.18 A Hough kernel that can compute $\rho(\theta)$ and $\rho(180° - \theta)$ using two multipliers and two adders. This architecture assumes $\theta_0 = 0°$, $\theta_{N_\theta/2} = 90°$, and N_θ is an even number. 74
3.19 An FPGA architecture of the Look-Ahead Kernel, which calculates $y\sin(\theta)$ after each image row and stores the results in N_θ registers. Control logic has been removed from this design to simplify the illustration. 75
3.20 An illustration of a vehicle equipped with a video camera to monitor the lane markings on each side of the vehicle. 80
3.21 A digital image of road markings to be processed using the LHT (a). The detected lines are reconstructed and overlaid on top of the image (b). 81
3.22 An illustration of a UAV equipped with a digital video camera that can perform automated tasks in several application areas, including power line analysis, crop line analysis, wind turbine tracking, and railroad evaluation. 82
3.23 A digital image of a building crack to be processed using the LHT (a). The detected lines are reconstructed and overlaid on top of the image (b). 83
3.24 An illustration of an industrial vision system to inspect products. This particular diagram depicts a camera directed at a conveyor belt of food products. 84
3.25 A digital image of a cracked biscuit to be processed by the LHT (a). The detected line is reconstructed and overlaid on top of the biscuit image (b). Note that the line thickness has been increased in size for inspection purposes. 85
3.26 The LHT is applied to an image of a QPSK constellation (a) and 16-QAM constellation (b), resulting in an HPS for each example given in (c) and (d), respectively. 86

4.1 System overview of the HEP that targets the Zynq MPSoC. The PL accelerates the user's custom LHT architecture, while the PS plots and evaluates the resulting HPS. 93

4.2 Detailed diagram of the HIU, consisting of an AXI DMA, the HPA, AXI4-Stream Broadcasters, and the user's custom architecture. 95

4.3 FPGA architecture of the HPA. The system contains two AXI4-Stream interfaces, three AXI4-Lite registers, control circuitry, an HDL counter, and a relational operator. 98

4.4 HPS visualisation using `go.surface` (a) and `go.heatmap` (b) interactive plotting APIs. The user can interact with the plots by zooming, panning, and hovering their mouse cursor over the plot to reveal cell data. 100

4.5 The IP Integrator block design for the HEP *HDL Coder* reference design. 108

4.6 A UML diagram representing the properties, methods, and classes of the HEP software drivers and APIs. 110

4.7 A diagram depicting the HEP embedded system design workflow. 112

4.8 The Simulink reference design template to simplify HEP integration. . . 113

4.9 A screenshot of the *HDL Coder* Workflow Advisor. 114

4.10 A screenshot of the HDL Workflow Advisor during Step 1.2. The board IP address is set for SSH communication. 115

4.11 A screenshot of the HDL Workflow Advisor during Step 1.4. The system clock frequency is configured. 115

4.12 A screenshot of the HDL Workflow Advisor during Step 1.3. Setting the target interfaces for HDL code generation. 116

4.13 A screenshot of the Jupyter environment that is accessed using a web browser. The HEP software drivers, bitstream, and default notebook have been transferred into the file system using SSH. 116

4.14 Two test images (a) and (b). Their corresponding edge images (c) and (d) using Sobel edge detection with a threshold of 80 and 70, respectively. 120

4.15 HPS results for the hardware validation of the parallel LHT architecture on the XCZU7EV-2E device. The isometric view of the HPS for the window image (a), and the stairs image (b). The top-down view of the HPS for the window image (c), and the stairs image (d). 121

4.16 Line reconstruction results for the test images input into the parallel LHT architecture. The reconstructed line images of the window (a) and stairs (b). The reconstructed lines overlaid on top of the original colour images of the window (c) and stairs (d). 122
4.17 This screenshot presents the HEP Jupyter environment, where the LHT architecture is undergoing hardware validation using the window image. 123
4.18 This screenshot presents the HEP Jupyter environment, where the LHT architecture is undergoing hardware validation using the stairs image. . 124

5.1 Functional block diagram of the Symmetric LHT, which presents five main stages of the algorithm. 131
5.2 An FPGA architecture that multiplies two input signals by a constant value. 132
5.3 An FPGA architecture that exploits resource sharing to multiply two input signals by a constant value using one multiplier. 132
5.4 Coordinate system used by the parallel LHT architecture (left). Coordinate system used by the Symmetric LHT architecture (right). 134
5.5 Scanning an $M \times N$ image (left), parallel pixel stream of the image (right). Each row of the image is streamed from the image centre to the image borders. 136
5.6 System overview of the Symmetric LHT architecture design. 142
5.7 The pixel packing architecture that converts from raster streaming format to the parallel pixel streaming format described in Section 5.2.3. . . 143
5.8 FPGA architecture for a simple coordinate calculator. This design uses two fixed-point counters; one counter tracks the x position, and another counter tracks the y position. 144
5.9 A simple phase controller design for the Symmetric LHT. The phase controller generates several signals for resource sharing operations in the symmetric Hough kernel. 145
5.10 Generalised architecture for the symmetric Hough kernel. 146

5.11 FPGA architecture for the top-level accumulator design. The accumulation controller implements an FSM that maintains the operation of the accumulation memory. Control logic has been removed from this design to simplify the illustration. 148

5.12 FSM diagram for the accumulator controller. The FSM contains four states: IDLE, VOTE, READ, and CLEAR. The default entry state is IDLE. If an undefined state occurs, the FSM returns to the IDLE state. 149

5.13 Overview diagram of the accumulator memory subsystem. Notice that there are $N_\theta/2 - 1$ bit-packed accumulator memories. 153

5.14 The bit-packed accumulator memory architecture design that performs voting for two angles in θ. The BRAM selected is configured in SDP mode, which allows one read and write transaction per clock cycle. The accumulation path is indicated by the dashed line. 154

5.15 A general circuit diagram for the θ_0 and $\theta_{N_\theta}/2$ Accumulators. The accumulation path is indicated by the dashed line. 155

5.16 HPS results for the hardware validation of the Symmetric LHT architecture on the XCZU7EV-2E device. The isometric view of the HPS for the window image (a), and the stairs image (b). The top-down view of the HPS for the window image (c), and the stairs image (d). 159

5.17 Line reconstruction results for the test images input into the Symmetric LHT architecture. The reconstructed line images of the window (a) and stairs (b). The reconstructed lines overlaid on top of the original colour images of the window (c) and stairs (d). 160

5.18 A bar chart comparing the number of memory bits required by each LHT algorithm across several image resolutions. 165

6.1 Functional block diagram of the ARLHT algorithm. 168

6.2 The HPS used by the Gradient LHT (a) contains peaks that are sparser than the HPS used by the ARLHT (b). 169

6.3 The RBM of the chessboard image using the ARLHT algorithm. Notice that only one memory array is required to store the RBM for the ARLHT. 170

6.4 The ARLHT RBM with morphological opening is presented in (a). The decompressed HPS of the chessboard is given in (b). The vote threshold used to generate the decompressed HPS is 60% of the maximum vote. . 172
6.5 Line reconstruction results for the ARLHT algorithm. 173
6.6 System overview of the ARLHT architecture design. 174
6.7 FPGA architecture of the ARLHT's Hough kernel. The output $\rho(\alpha)$ value is incremented by $D/2$ for memory addressing purposes. 175
6.8 FPGA architecture for calculating the adjusted orientation when $K_\theta = 4$. As highlighted in the red box, there are $K_\theta - 1$ adjustment stages in total (one initial adjustment stage, and two regular adjustment stages). . . . 176
6.9 FPGA architecture for the ARLHT's top-level accumulator design. The accumulation controller implements an FSM to control the accumulator memory. Note that the TLAST output is only valid when the clear output of the accumulator controller is high. 177
6.10 A circuit diagram of the ARLHT accumulator memory subsystem. The accumulation path is indicated by the dashed line. 178
6.11 FSM diagram for the ARLHT's accumulator controller. 179
6.12 FPGA architecture for the ARLHT's top-level RBM circuit design. The RBM controller implements an FSM to control the RBM memory. . . . 182
6.13 FSM diagram for the ARLHT's RBM controller. 183
6.14 The RBM memory subsystem architecture that performs RBM voting. The BRAM selected is configured in SDP mode. 185
6.15 Peak separation and detection architecture for the ARLHT. 185
6.16 HPS results for the hardware validation of the ARLHT architecture on the XCZU7EV-2E device. The isometric view of the HPS for the window image (a), and the stairs image (b). The top-down view of the HPS for the window image (c), and the stairs image (d). 189

6.17 Line reconstruction results for the test images input into the ARLHT architecture. The reconstructed line images of the window (a) and stairs (b). The reconstructed lines overlaid on top of the original colour images of the window (c) and stairs (d). 190

6.18 A bar chart comparing the number of memory bits required by the parallel LHT, Symmetric LHT, standard LHT, and ARLHT across several image resolutions. 193

A.1 An example of rotating a vector (x_0, y_0) three times on the Cartesian plane. 204

A.2 Plots of the first three CORDIC iterations using the rotation angles given in Table A.1. The rotation angle is given to one decimal place and becomes smaller after each iteration. The scaling factor (discussed in Section A.2) is ignored. 206

A.3 These plots demonstrate vector growth after three CORDIC iterations. The rotation angles are given to one decimal place and the vector length is given to four decimal places. 207

A.4 A plot demonstrating the first six CORDIC rotations tending towards a limit in the anticlockwise and clockwise directions. 209

A.5 Two plots demonstrating the rotation of a vector into the region of convergence for Circular CORDIC. The left plot illustrates a vector rotation by 90° in the clockwise direction, while the right plot shows a vector rotation by 90° in the anticlockwise direction. 210

A.6 An example of Circular CORDIC operating in vectoring mode. The input vector is rotated towards the x-axis. 211

A.7 Hardware architecture of Circular CORDIC operating in vectoring mode. 212

A.8 A hardware architecture of a Circular CORDIC cell for vectoring mode operation. 213

B.1 This diagram presents a subset of MathWorks *HDL Coder* blocks used to develop the LHT architecture designs presented in this thesis. 216

B.2 A diagram illustrating the Simulink convention for displaying a fixed-point data type and dimension of a signal. 218
B.3 An overview diagram of the parallel LHT architecture and HEP infrastructure. 218
B.4 A Simulink system presenting the contents of the DUT block, which contains the parallel LHT kernel and accumulator. The *convert* block is used to ensure the TDATA signal at the output of the DUT is 32 bits, which is compliant with the HEP design methodology. 219
B.5 The contents of the *parallel_lht_kernel* Simulink subsystem. 220
B.6 The Simulink system for the *coordinate_counters* block. 220
B.7 The *parallel_kernel* Simulink subsystem. 221
B.8 Simulink model of the Look-Ahead Kernel, which is detailed in [20]. The design uses a single multiplier to pre-compute $y\sin(\theta)$ for the next image row. A tapped-register is employed to write values to an output register for use in the *rho_computation* subsystem. 221
B.9 The contents of the *rho_computation* subsystem, which implements (3.2) using the technique described in [87]. 222
B.10 The Simulink diagram of the *parallel_lht_accumulator* subsystem. 222
B.11 The *accumulator_array* Simulink design. 226
B.12 The contents of the *angle_accumulator* Simulink subsystem. 227

C.1 This screenshot presents the HEP Jupyter environment, where the Symmetric LHT architecture is undergoing hardware validation using the window image. 231
C.2 This screenshot presents the HEP Jupyter environment, where the Symmetric LHT architecture is undergoing hardware validation using the stairs image. 232

D.1 This screenshot presents the HEP Jupyter environment, where the ARLHT architecture is undergoing hardware validation using the window image. 234

D.2 This screenshot presents the HEP Jupyter environment, where the ARLHT architecture is undergoing hardware validation using the stairs image. . 235

List of Tables

2.1 36 Kb BRAM memory schemes. 17

2.2 18 Kb BRAM memory schemes. 17

2.3 FPGA resources available on the XCZ7UEV-2E device. 31

4.1 FPGA resource requirements for the HEP on the XCZ7UEV-2E device. 117

4.2 FPGA resource requirements for the HEP and LHT on the XCZ7UEV-2E device. 117

4.3 The HPS memory consumption of the standard LHT and the parallel LHT. 118

4.4 Processing time results of parallel LHT architectures that target various image resolutions. The processing time column contains measurement results from the HEP. 119

5.1 LHT results for two points A and B, when $\delta_\theta = 22.5°$. 135

5.2 Platform agnostic memory allocation table for the bit-packed accumulator. 137

5.3 BRAM allocation table for a memory instance of the bit-packed accumulator when $N_\rho = 2204$, $N_\theta = 180$, and $b = 12$ bits. 139

5.4 An efficient BRAM allocation table for a memory instance of the bit-packed accumulator when $N_\rho = 2204$, $N_\theta = 180$, and $b = 12$ bits. 140

5.5 Accumulator controller output values. 150

5.6 Accumulator controller configuration for the Current State register. . . . 151

5.7 Accumulator controller configuration for the ρ_{count} and θ_{count} registers. A 'No Condition' entry in the condition column indicates that the transition between the current value and the next value always occurs. . . . 151
5.8 FPGA resource consumption for the Symmetric LHT architecture (without the HEP), which is implemented on the XCZU7EV-2E device. . . . 156
5.9 The memory consumption of the parallel LHT and the Symmetric LHT for various image resolutions. . . . 157
5.10 Processing time results of Symmetric LHT architectures that target various image resolutions. The Processing Time column contains measurement results from the HEP. . . . 158
5.11 Results of the Symmetric LHT and comparison with related works. . . . 162
5.12 The memory consumption of the standard LHT and Symmetric LHT. . 164

6.1 Accumulator controller output values for the ARLHT. . . . 180
6.2 Accumulator controller Current State register for the ARLHT. . . . 181
6.3 Accumulator controller ρ_{count}, θ_{count}, and K_{count} registers for the ARLHT. 181
6.4 RBM controller output values for the ARLHT. . . . 183
6.5 RBM controller Current State register for the ARLHT. . . . 184
6.6 RBM controller ρ_{count} and θ_{count} registers. . . . 184
6.7 FPGA resource consumption for the ARLHT architecture, which is implemented on the XCZU7EV-2E device. . . . 186
6.8 The memory consumption of the standard LHT and the ARLHT architecture. . . . 187
6.9 Processing time results of ARLHT architectures that target various image resolutions. The processing time column contains measurement results from the HEP. . . . 188
6.10 Results of the ARLHT and comparison with related works. . . . 191

A.1 The rotation angles (degrees) for the first four iterations of the CORDIC algorithm. The rotation angles are given to four decimal places. . . . 206

A.2 The vector growth for the first four iterations of the CORDIC algorithm. The rotation angles (degrees) and vector growth are given to four decimal places. 208

A.3 Example of Circular CORDIC operating in vectoring mode using eight iterations. Each value is given to four decimal places. 214

List of Acronyms

ADAS Advanced Driver Assistance Systems

AHT Adaptive Hough Transform

AMBA Advanced Microcontroller Bus Architecture

AMD Advanced Micro Devices

API Application Programming Interface

APU Application Processor Unit

ARLHT Angular Regions LHT

ASIC Application-Specific Integrated Circuit

AXI4 Advanced eXtensible Interface 4

BRAM Block RAM

CHT Circle Hough Transform

CLB Configurable Logic Block

CNN Convolutional Neural Network

CORDIC COordinate Rotation DIgital Computer

CUDA Compute Unified Device Architecture

DA Distributed Arithmetic

DDR Double Data Rate

DM Distributed Memory

DMA Direct Memory Access

DNN Deep Neural Network

DSP Digital Signal Processing

DUT Design Under Test

FF Flip-Flop

FFT Fast-Fourier Transform

FHD Full High Definition

FIFO First In First Out

FIHT2 Fast Incremental Hough Transform 2

FIL FPGA-in-the-loop

FILO First In Last Out

FIR Finite Impulse Response

FPGA Field Programmable Gate Array

fps frames per second

FSM Finite State Machine

GPU Graphics Processing Unit

HDL Hardware Description Language

HDMI High-Definition Multimedia Interface

HEP Hough Evaluation Platform

HIU Hough Inspection Unit

HLS High Level Synthesis

HPA Hough Performance Analyser

HPS Hough Parameter Space

HT Hough Transform

IP Intellectual Property

IP Internet Protocol

KHT Kernel Hough Transform

LFMCW Linear Frequency-Modulated Continuous Waveform

LHT Line Hough Transform

LUT Lookup Table

MFR Modulation Format Recognition

MPSoC Multi-Processor System on Chip

OpenCL Open Computer Language

OpenCV Open Computer Vision

OWC Optical Wireless Communication

PCB Printed Circuit Board

PHT Probabilistic Hough Transform

PL Programmable Logic

PPHT Progressive Probabilistic Hough Transform

PS Processing System

PYNQ Python Productivity for ZYNQ

QAM Quadrature Amplitude Modulation

QPSK Quadrature Phase Shift Keying

RAM Random Access Memory

RBM Region Bitmap

RFSoC Radio Frequency System on Chip

ROI Region of Interest

ROM Read Only Memory

RPU Real-Time Processor Unit

SDK Software Development Kit

SDP Simple Dual-Port

SNR Signal-to-Noise Ratio

SoC System on Chip

SoF Start of Frame

SPI Shared Peripheral Interrupt

SSH Secure Shell

SSR Super Sample Rate

Tcl Tool Command Language

TDP True Dual-Port

UAV Unmanned Aerial Vehicle

UML Unified Modeling Language

VCU Video Codec Unit

VHDL Very High Speed Integrated Circuit HDL

VLNV vendor, library, name, and version

XDC Xilinx Design Constraints

Chapter 1

Introduction

1.1 Research Background

Vision is an important part of human perception and plays an essential role in our daily lives. Therefore, it is unsurprising that computer vision systems have emerged to carry out visual inspection tasks [1]. Typical computer vision systems consist of many components each applying their own algorithms and decision making processes. These systems are the foundation of all visual inspection applications and allow a computer to see and interpret its environment.

A significant challenge for the design of modern vision systems is to accommodate ever-increasing image resolutions and video frame rates. For example, the Full High Definition (FHD) video standard, which has an image resolution of 1920×1080 pixels and displays 60 frames per second (fps), has a throughput of over 124 million pixels per second. To meet these demands, improved processing performance and algorithm optimisation is essential. This issue is exacerbated when considering applications that have real-time constraints and are computationally complex. These tasks require specialised hardware such as an Application-Specific Integrated Circuit (ASIC), Graphics Processing Unit (GPU), or a Field Programmable Gate Array (FPGA). Due to their availability, reprogrammable architecture, cost, and low power consumption, FPGAs are commonly selected for embedded vision applications [2].

Line extraction in digital images is essential for many embedded vision tasks. It is a difficult problem to achieve in the spatial image domain, as it is necessary to globally determine collinearity between image feature points. However, it is possible to reduce the complexity of line detection by converting the entire image of spatially positioned feature points, to an associated parameter space. This procedure is formally known as the Line Hough Transform (LHT) [3]. The LHT is robust to extraction under conditions where a line is partially occluded or deformed. Furthermore, it is tolerant of noise, and unlike the Trace Transform [4], of which the LHT is a variation, it is also capable of extracting multiple lines in a single pass of a digital image.

The LHT is a useful tool in many application areas. It is commonly used by lane departure warning systems for safely guiding vehicles [5] and is a popular choice for processing digital images that are captured from Unmanned Aerial Vehicles (UAVs). Recent applications that involve UAVs include power line detection [6] and railroad surveillance [7]. The LHT is also used in wireless communication systems. In particular, there have been investigations of using the LHT to classify modulation schemes in Optical Wireless Communications (OWCs) [8]. The LHT has also been used to assist in the detection of Linear Frequency-Modulated Continuous Waveforms (LFMCWs) in low Signal-to-Noise Ratio (SNR) environments [9].

Although the LHT reduces the complexity of extracting lines from a digital image, there are several disadvantages of using the algorithm. The LHT is computationally demanding, which may result in slow processing times. There is also a high resource cost associated with maintaining the Hough Parameter Space (HPS) in memory, as it represents all possible parameter combinations for detected features in a digital image, leading to high storage requirements. There have been several different adaptations of the LHT that improve processing time, accuracy, and resource cost of the algorithm [10–12]. Many studies have implemented the LHT on an FPGA to improve processing performance [13–15]. A significant aspect that is often overlooked is the memory consumption of the HPS. The resources required often saturate dedicated FPGA memories, which becomes even more of an issue when using larger image resolutions as the memory consumption of the HPS increases.

Semiconductor companies, such as Advanced Micro Devices (AMD) [16] and Intel [17], offer FPGA devices in different sizes, which they usually classify as small, mid-range, or high [18, 19]. At the time of writing, small FPGAs typically have up to a few thousand logic elements, mid-range FPGAs consist of tens of thousands of logic elements, and large FPGAs can contain hundreds of thousands of logic elements. Researchers and developers in previously published literature commonly implement the LHT on mid-range and large FPGA devices as they offer a suitable amount of on-chip memory resources to store the HPS [20–22]. However, using large and mid-range FPGA devices for embedded vision algorithms, such as the LHT, can be financially expensive, especially if the FPGA device is used for mass manufacture. Implementing algorithms on small FPGAs can improve the overall cost of the final solution, which is a crucial factor for budget-constrained projects and high-volume production.

There are very few studies that optimise or reduce the memory requirements of the LHT [12, 23]. Furthermore, there has yet to be an investigation into the use of compression and architectural design to optimise the memory efficiency of the LHT for implementation on small FPGA devices. FPGAs have limited memory capabilities and require innovative techniques to decrease the memory requirements of the HPS and reduce the financial cost of the final solution. Due to its widespread adoption and robust extraction of lines, optimising the memory efficiency of the LHT on an FPGA is a challenging research problem that requires fundamental research to solve.

1.2 Research Aims and Objectives

The work in this thesis aims to reduce the memory requirements of the LHT for FPGAs by developing new algorithms and architectures that improve memory consumption. In particular, the proposed designs use architectural and algorithmic optimisations to reduce the memory requirements of the HPS. The research goals of this work are separated into three main objectives, which are listed below.

1. The first objective is to develop an evaluation platform for LHT architectures, which will be used to validate hardware designs and compute their processing

time. An LHT architecture should also be developed to test the functionality of the evaluation platform. Later, when developing memory-efficient LHT architectures, the evaluation platform can be used for design validation.

2. Objective two is to design a memory-efficient LHT that uses novel architectural optimisations to reduce memory consumption in FPGA devices.

3. The final objective is to optimise the memory consumption of the HPS by modifying the LHT algorithm, which may affect the accuracy and robustness of line extraction.

1.3 Original Contributions

The research undertaken in this thesis has resulted in several original contributions to knowledge. These contributions are summarised as follows.

1. Previously reported LHT architecture designs often use software simulations for architecture validation and do not provide any testing criteria. In Chapter 4, a novel evaluation platform for designing and validating LHT architectures is presented. The system targets the AMD Zynq Multi-Processor System on Chip (MPSoC) device [24] and uses the Python Productivity for ZYNQ (PYNQ) framework [25] to support the visualisation and control of an FPGA design. The evaluation platform supports testing of an LHT architecture on the physical target device, which improves validation of a hardware architecture beyond that of software simulations. The platform can also accurately measure the time taken for an LHT architecture to process an image. It is also integrated into MathWorks *HDL Coder* workflows for rapid architecture development and system deployment [26]. The evaluation platform was first published in [27]. To the author's knowledge, this evaluation platform is the first reported design that combines MathWorks and PYNQ development tools and therefore represents an additional milestone in System on Chip (SoC) systems development.

2. Chapter 5 presents a novel technique to reduce arithmetic and memory resources to compute the LHT using AMD FPGAs. A symmetric Hough kernel and bit-packed accumulator memory is proposed that are optimised to reduce resource computation and decrease on-chip memory allocation. This technique is known as the Symmetric LHT, as the algorithm exploits symmetry in the spatial image domain. It produces similar line detection results as the LHT described in [3] and is the first technique to leverage spatial domain symmetry and bit-packing techniques to improve the memory consumption of the HPS.

3. An FPGA architecture of the Symmetric LHT is also described in Chapter 5. The architecture is designed for an image consisting of 1920×1080 pixels and is developed and validated using the novel evaluation platform presented in Chapter 4. The Symmetric LHT reduces the arithmetic resource requirements and decreases on-chip memory consumption in comparison to previously published work. The Symmetric LHT architecture design employs a memory bit-packing scheme, which does not affect the accuracy of line detection. Unlike previously published works, the Symmetric LHT architecture can target small FPGAs across various image resolutions, achieving a low-cost, embedded solution for applications requiring dedicated line detection in digital images.

4. A lossy compression scheme for reducing the memory requirements of the LHT is presented in Chapter 6 of this thesis. The memory-efficient algorithm is named the Angular Regions LHT (ARLHT). It was recognised that the work presented in [23], wherein the HPS was reduced in size by partitioning the input image into subregions, could be further improved in terms of memory utilisation. As such, the ARLHT improves the compression of the HPS by dividing the input image into angular regions. The ARLHT is the first algorithm designed for FPGA implementation that exploits compression to reduce the memory requirements of the HPS.

5. An FPGA architecture of the ARLHT algorithm is presented in Chapter 6, wherein the architecture is carefully designed to prevent unnecessary allocation of

FPGA resources. The architecture design is developed using the novel evaluation platform described earlier, and the FPGA resource requirements are analysed. For a 1920×1080 pixel image, the ARLHT architecture consumes fewer FPGA arithmetic and memory resources than the LHT presented in [3] and other previously published implementations of the LHT. As such, the ARLHT architecture can process FHD video on small, low-cost FPGA devices.

1.4 Publications and Awards

Various publications were created during the author's studies, which are listed below. The publications directly related to this research are (a) and (b). The publications (c)-(e) are related to the author's studies on the Zynq Radio Frequency System on Chip (RFSoC), Zynq MPSoC, and Zynq SoC. These studies are relevant to the work in this thesis as they use similar design methodologies and devices.

(a) [27] D. Northcote, L. H. Crockett, P. Murray and R. W. Stewart, "A PYNQ evaluation platform for FPGA architectures of the Line Hough Transform", in *IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, Springfield, MA, USA, Sep 2020, pp. 133-137.

Electronic Access: Conference Paper | Video Presentation | Slides

(b) [28] D. Northcote, L. H. Crockett and P. Murray, "FPGA implementation of a memory-efficient Hough parameter space for the detection of lines", in *IEEE 50th International Symposium on Circuits and Systems (ISCAS)*, Florence, Italy, May 2018, pp. 1-5.

Electronic Access: Conference Paper | Video Presentation | Slides | Poster | Award

(c) [29] J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework", *IEEE Access*, vol 8, pp. 129012-129031, Jul 2020.

Electronic Access: Journal Paper

(d) [30] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson and R. W. Stewart, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*, Strathclyde Academic Media, Apr 2019.

Electronic Access: Textbook Download

(e) [31] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and D. Northcote, *The Zynq Book Tutorials for Zybo and Zedboard*, Strathclyde Academic Media, Aug 2015.

Electronic Access: Textbook Download

The author's research in (b) received international recognition upon being awarded the best student paper at the IEEE ISCAS conference.

1.5 Thesis Structure

The remainder of this thesis is laid out as follows. In Chapter 2, a brief review of FPGA and SoC technology is provided and relevant design and development tools are introduced. Subsequently, FPGA and SoC vision architectures associated with the work in this thesis are described. This includes an overview of hardware accelerated image processing, local image filtering operations, edge detection, and binary morphological processing. Relevant FPGA architectures are presented throughout when appropriate.

Chapter 3 describes the LHT algorithm and discusses vote accumulation and line reconstruction. A literature review on the LHT is provided, which primarily explores software design and optimisations, multiplierless FPGA architectures, resource-efficient FPGA architectures, and various candidate applications that could benefit from using an embedded LHT design.

In Chapter 4, the Hough Evaluation Platform (HEP) that enables the design and validation of LHT architectures is presented. In particular, the system specification, system design, visualisation features, and rapid integration capabilities are explored. FPGA resource analysis and testing of the HEP using an LHT architecture (described in Appendix B) is provided.

The Symmetric LHT is presented in Chapter 5. Important concepts regarding the design of the Symmetric LHT are described, including spatial domain symmetry, parallel pixel processing, and memory bit-packing. The architecture of the Symmetric LHT is then presented, which includes its pixel packing system, symmetric Hough kernel, accumulation controller, and bit-packed accumulator. FPGA implementation results and processing time analysis as computed by the HEP are reported.

Chapter 6 presents the ARLHT algorithm and describes the memory compressed HPS, the adopted voting scheme, and the associated line extraction technique. A simple example of the ARLHT operating on an image is presented, where line reconstruction is applied to demonstrate results. The ARLHT architecture design is then described, which includes its preprocessing and segmentation architecture, resource-efficient Hough kernel, accumulation arrays, and line extraction design. The resource consumption of the FPGA architecture is reported and the HEP is used to evaluate its processing time.

Finally, conclusions and future research opportunities are discussed in Chapter 7.

Chapter 2

Embedded Vision Systems

2.1 Introduction

A computer that forms part of a more extensive system is known as an embedded system and is usually configured to execute a specific task [32]. For example, a factory manufacturing line may contain an embedded vision system that is configured to inspect product defects. Many embedded systems use FPGAs to accelerate algorithms and tasks by exploiting their parallel processing capabilities. FPGAs can be reprogrammed after deployment, making them particularly suitable for flexible vision applications. With the advent of heterogeneous processing devices, such as the AMD Zynq MPSoC which combines an FPGA with a multi-element processing system, embedded systems are becoming increasingly adaptable. The Zynq MPSoC is capable of achieving flexible vision applications through dynamic hardware reconfiguration with multiprocessor capabilities.

This chapter initially discusses and compares GPUs, ASICs, and FPGAs as candidate technologies for implementing flexible vision applications. An overview of FPGA technology and its specialised resources is then presented. The Zynq MPSoC is also described, and the principles of data movement between its constituent processing elements are discussed. The PYNQ framework is also introduced, and its advantages towards Zynq-based development and architecture validation are highlighted. Finally, embedded vision systems using FPGA and Zynq MPSoC technologies are explored.

Relevant image processing algorithms and their FPGA architectures are also described. These include local filtering, edge detection, and binary morphology. Each of these techniques and algorithms support the FPGA architectures of the LHT described later in Chapters 5 and 6.

2.2 Embedded Hardware Platforms

GPUs, ASICs, and FPGAs are the most popular hardware platforms for implementing embedded vision systems due to their high computational performance. Each platform has strengths and weaknesses depending on application requirements. A summary of each platform and its capabilities relating to embedded vision applications is as follows:

1. GPU — Modern GPUs are well-suited for vision tasks that require high parallelism and data throughput, such as feature extraction using Convolutional Neural Networks (CNNs) [33], which are a type of deep learning algorithm. GPUs are programmed using languages such as Compute Unified Device Architecture (CUDA) [34] or Open Computer Language (OpenCL) [35]. These open-source languages provide a high level of design abstraction, making it easier to develop vision applications on GPUs than FPGAs and ASICs. However, studies report that GPUs are less energy-efficient and have higher latency than ASICs and FPGAs when performing specific vision tasks [2,36]. While GPUs are a popular choice for embedded vision tasks, their energy efficiency and latency depend on many factors, such as the available memory bandwidth, the image resolution, and the deployed algorithm.

2. ASIC — Applications that require high performance, low power consumption, low latency, and high-volume manufacturing are suited for ASICs. These custom-designed integrated circuits can accelerate video pipelines that perform a specific visual task, such as region of interest extraction in a production image sensor [37]. However, ASICs are expensive to manufacture and design, and development tools can cost over $1,000,000 for 28 nm technology [38]. GPUs and FPGAs do not typically incur these expenses, as many open-source development languages and

free-to-use design tools are available. It is necessary to redesign and manufacture an ASIC if the target application updates or changes, as they are not reprogrammable unlike FPGAs and GPUs.

3. FPGAs — Vision applications that require high performance and flexibility can leverage FPGAs, as they provide parallel processing capabilities and the option to reprogram a hardware design in response to application updates or changes. Due to their reprogrammable facilities, FPGAs are useful for rapid prototyping and developing digital circuits and systems. While FPGAs are generally more energy efficient than GPUs, they are less energy efficient than ASICs. Additionally, high-end FPGAs can be very expensive compared to GPUs and ASICs.

The choice of an embedded hardware platform depends on the specific requirements of the target application. The work in this thesis focuses on prototyping efficient LHT architecture designs, for which FPGAs are the candidate solution. FPGAs offer high computational performance and low latency processing, which is advantageous for the candidate applications presented in Section 3.4. These applications include lane detection for vehicles, modulation format recognition in optical wireless communications, and product defect inspection on manufacturing lines.

2.3 Overview of FPGA Technology

An FPGA is an integrated circuit populated by low-level logic elements and interconnects that are user-programmable after manufacture. Due to their reprogrammable capabilities, FPGAs are suited for rapid prototyping, iterative research and development, and embedded applications that exploit reconfigurable hardware. FPGAs are generally low-cost, widely available, and have highly effective parallel processing capabilities that suit Digital Signal Processing (DSP) algorithm implementation.

This section reviews the fundamentals of FPGA technology to gain an appreciation for FPGA system design and development. In particular, descriptions and illustrations are based on the AMD Kintex UltraScale+ device family [39], as this contains the target FPGA logic fabric in this thesis and it is used in the programmable logic portion

of the Zynq MPSoC device (discussed further in Section 2.4). Additionally, the FPGA development process and design tools are also described. An overview of the Kintex UltraScale+ FPGA architecture is shown in Figure 2.1.

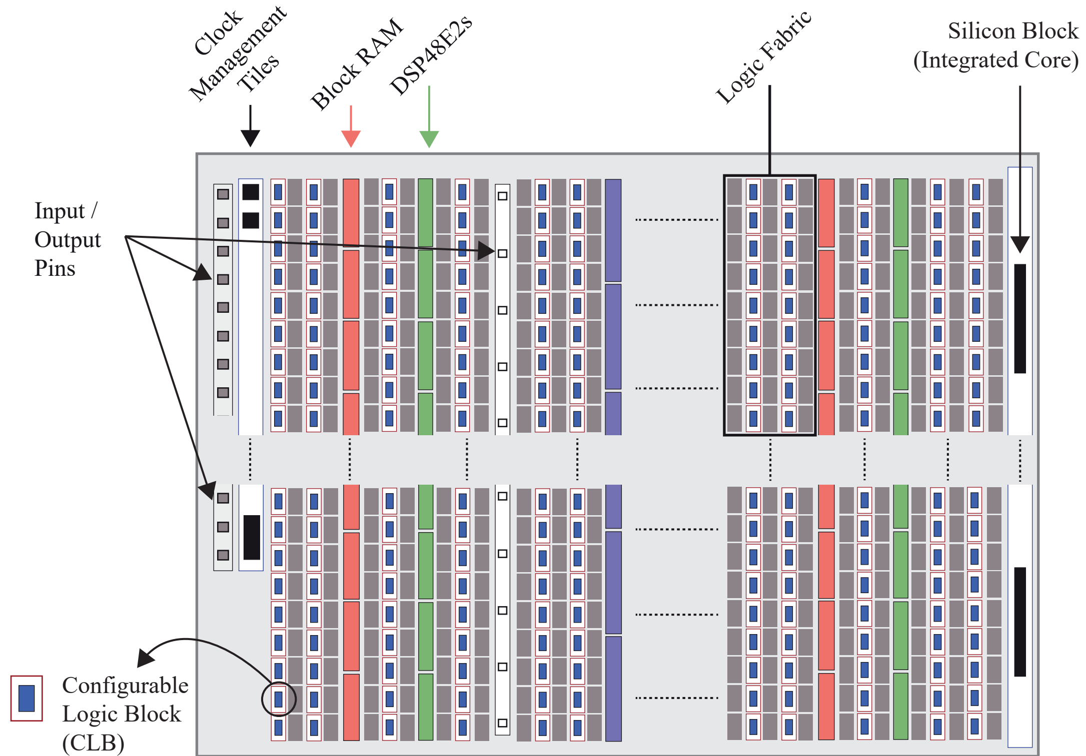


Figure 2.1: An overview of the AMD Kintex UltraScale+ FPGA architecture.

2.3.1 The Logic Fabric

FPGAs consist of many Configurable Logic Blocks (CLBs) [40] that are placed in a two-dimensional array. As illustrated in Figure 2.2, each CLB contains one slice that is host to several logic elements for implementing user-defined circuits. For Kintex UltraScale+ devices, a slice contains 8 $\times$ 6-input Lookup Tables (LUTs), Flip-Flops (FFs), and other logic to support signal routing. Vertically neighbouring CLBs can be chained together using carry-logic, which includes multiplexers and logic chain connections. Connecting many CLBs together allows larger logic circuits to be implemented. A switch matrix is also positioned adjacent to each CLB to enable communication with other FPGA resources.

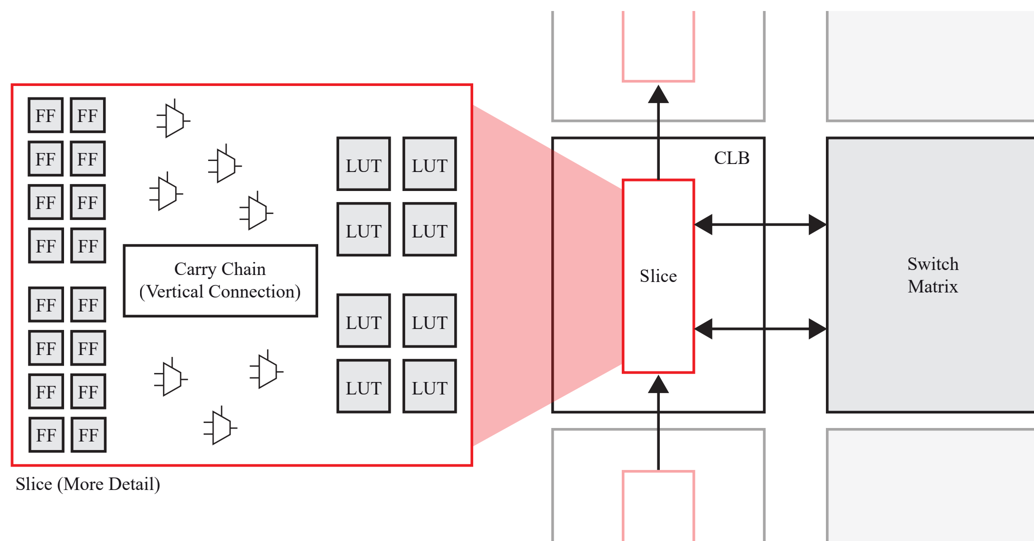


Figure 2.2: An illustration of a CLB and switch matrix. The logic elements contained inside a slice are also shown.

The CLBs and switch matrices within an FPGA are collectively referred to as the logic fabric. The logic fabric can be routed and configured to implement arithmetic and memory circuits. Common mathematical functions such as addition, multiplication, and division can be constructed using several CLBs. There is also a well-known technique named Distributed Arithmetic (DA) that can efficiently implement sum-of-product operations using LUTs rather than multipliers [41]. The logic fabric can also be used to create small local memories known as Distributed Memory (DM). DM circuits exploit the storage facilities of LUTs and FFs, allowing FPGA designs to include Random Access Memory (RAM) and Read Only Memory (ROM) capabilities.

Modern FPGA devices also include specialised arithmetic and storage resources, which are implemented using dedicated silicon. In particular, the Kintex UltraScale+ family includes DSP48E2 slices and Block RAMs (BRAMs) for processing and storing data, respectively. These specialised resources are low-power, compact, and operate at high clock frequencies. They are arranged in columns throughout the FPGA logic fabric, allowing for simple routing to neighbouring components and logic elements. The remainder of this section describes the architecture and features offered by each specialised resource in detail.

2.3.2 The DSP48E2 Slice

The FPGA logic fabric can implement DA circuits using LUTs rather than multipliers. The demand for LUTs increase as arithmetic wordlengths grow longer. As illustrated in Figure 2.1 on page 12, the DSP48E2 slice [42] is a hard silicon component arranged in columns throughout the FPGA logic fabric. It was introduced to reduce the demand on FPGA fabric resources and improve arithmetic processing speed. DSP48E2 slices are suitable for implementing Finite Impulse Response (FIR) filters [43] and Fast-Fourier Transform (FFT) [44] architectures. A simplified architecture of the DSP48E2 slice is shown in Figure 2.3.

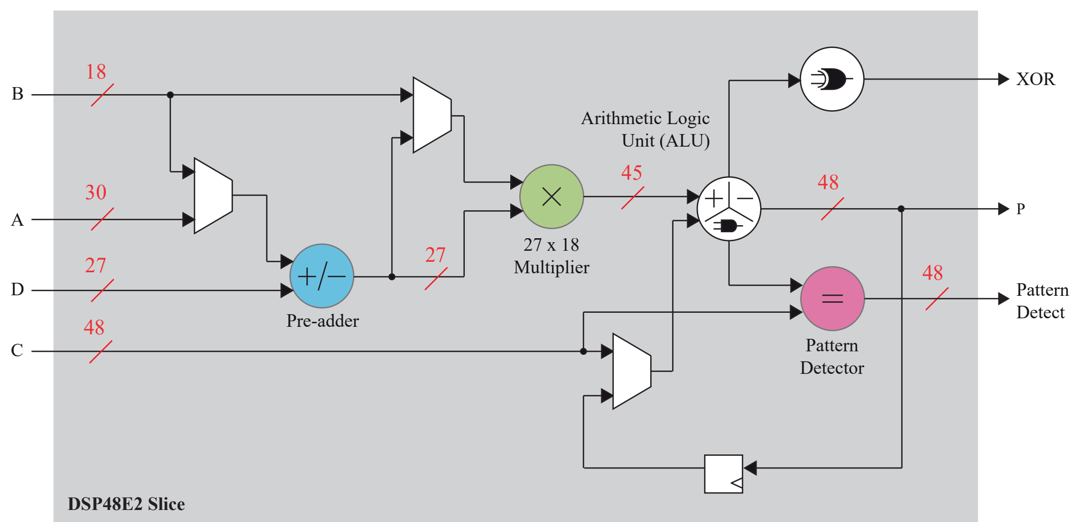


Figure 2.3: Simplified architecture of the DSP48E2 slice.

The DSP48E2 slice is capable of addition, subtraction, multiplication and bitwise logic manipulation. On close examination of Figure 2.3, one DSP48E2 slice can achieve the following fixed-point arithmetic operations:

- 30 bit $\times$ 27 bit pre-addition / subtraction,
- 27 bit $\times$ 18 bit multiplication,
- 48 bit $\times$ 45 bit addition / subtraction / bitwise logic operation.

Where longer wordlengths are required, it is possible to cascade two or more DSP48E2 slices together without using the FPGA logic fabric. For example, four DSP48E2 slices can be combined to create a 54 bit × 36 bit multiplication. The trade-off between arithmetic wordlengths and application performance must be carefully considered. Selecting unnecessarily long wordlengths can result in the inefficient allocation of DSP48E2 slices, subsequently increasing FPGA resource and power consumption. Similarly, selecting short wordlengths impacts the representable fixed-point range and quantisation effects, which affects arithmetic accuracy and precision.

DSP48E2 slice utilisation is considered for the FPGA architectures presented in Chapters 5 and 6. In each of these architectures, the primary aim when using DSP48E2 slices is to select suitable wordlengths to minimise the total allocation, while ensuring that accuracy and precision is acceptable for the given application.

2.3.3 Block RAM Tiles

RAMs, ROMs, and First In First Out (FIFO) buffers can be created using dedicated FPGA memories such as BRAM tiles [45]. In comparison to DM circuits, BRAMs are low-latency and can store up to 36 Kb of information per tile. They are arranged in columns throughout the FPGA logic fabric to provide neighbouring elements, such as DSP48E2 slices, with memory capabilities. As illustrated in Figure 2.4, it is possible to separate a 36 Kb BRAM into two individual 18 Kb BRAMs. Kintex UltraScale+ FPGAs use two BRAM primitives named RAMB36E2 and RAMB18E2. These primitives correspond to 36 Kb and 18 Kb BRAMs, respectively. Note that FPGA primitives are elementary components embedded in an FPGA and are configurable by the user [46].

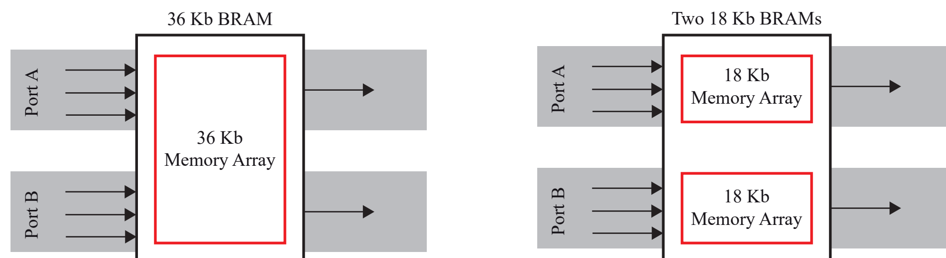


Figure 2.4: 36 Kb BRAM with dual-port interface (left), two 18 Kb BRAMs formed by separating a 36 Kb BRAM in half (right).

36 Kb and 18 Kb BRAMs can be configured to operate in True Dual-Ports (TDPs) mode and Simple Dual-Ports (SDPs) mode. Figure 2.5 presents a simplified diagram for TDP mode (left) and SDP mode (right) when using a RAMB36E2 primitive. In TDP mode, the BRAM primitive contains a dual-port interface, where each port can simultaneously read and write to memory. The data input ports are limited to 36 bits for RAMB36E2 primitives and 18 bits for RAMB18E2 primitives. There are also additional interfaces for each mode that allow two or more vertically neighbouring BRAMs to cascade and form large memory arrays.

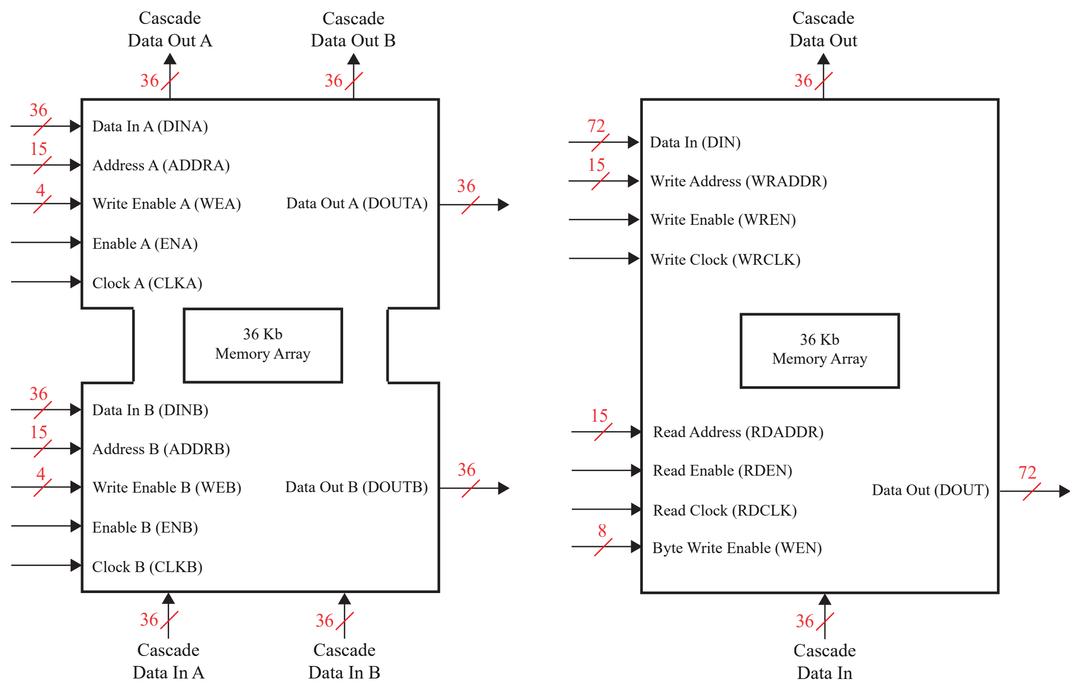


Figure 2.5: Simplified architecture of the RAMB36E2 primitive configured to operate in TDP mode (left), and SDP mode (right). Some ports have been removed or combined to simplify the diagram.

SDP mode is an alternative BRAM configuration. Each port of the BRAM's dual-port interface is combined in this configuration, allowing RAMB36E2 primitives to achieve data widths up to 72 bits, and RAMB18E2 primitives to achieve data widths up to 36 bits. It is only possible to perform one read and write operation in this configuration, as only one data input interface and one data output interface exist.

Memory resources can be efficiently allocated by configuring a BRAM's memory scheme. This technique is commonly referred to as 'reshaping' and is useful for adjusting the number of available addresses with respect to wordlength [45]. For example, a 36 Kb BRAM can be configured to store 1024 addresses × 36 bits, 2048 addresses × 18 bits, or other memory schemes as presented in Table 2.1.

Table 2.1: 36 Kb BRAM memory schemes.

No. Addresses	512 (SDP Only)	1024	2048	4096	8192	16384	32768
Wordlength (bits)	72 (SDP Only)	36	18	9	4	2	1

Similarly, 18 Kb BRAMs can be reshaped to optimise memory allocation as detailed in Table 2.2. Since 18 Kb BRAMs are smaller, there are fewer memory schemes available in comparison to 36 Kb BRAMs.

Table 2.2: 18 Kb BRAM memory schemes.

No. Addresses	512 (SDP Only)	1024	2048	4096	8192	16384
Wordlength (bits)	36 (SDP Only)	18	9	4	2	1

A BRAM's storage capacity, interface, and available memory schemes are significant properties that impact the design of an FPGA architecture and total on-chip memory consumption. These BRAM properties are important, as the FPGA architectures that are derived in Chapters 5 and 6 leverage BRAM tiles for storing the HPS.

2.3.4 Architecture Evaluation

Researchers and developers use an FPGA architecture's maximum achievable clock speed and total resource consumption to evaluate its operating performance in terms of processing speed and resource efficiency. FPGA architectures are typically compared against one another using these factors to determine the most effective design, i.e. the architecture that achieves the highest clock speed and/or lowest resource consumption. The remainder of this section describes an FPGA architecture's clock speed and resource consumption. These factors are used in Chapters 4, 5, and 6 to compare different FPGA architectures of the LHT.

Clock Speed

An FPGA architecture's maximum achievable clock frequency primarily depends on its underlying circuit design. One factor limiting the maximum clock frequency of an architecture design is its critical path, which is the combinatorial path between two clocked registers that exhibits the longest signal propagation delay. This path limits the maximum achievable clock frequency of the architecture design. Researchers and developers often evaluate an FPGA architecture using a target clock frequency, as computing its maximum achievable clock frequency is a time-consuming process.

Resource Consumption

Researchers commonly evaluate FPGA architectures for their resource consumption, which depends on several factors, such as the design complexity and the underlying algorithm. The resource consumption of an architecture is an important design consideration as it determines the size of the FPGA used, which influences the financial cost of the system. Architecture designs that use a considerable amount of resources require larger FPGAs, which are more expensive. Researchers often compare FPGA architectures for their consumption of LUTs, FFs, DSP slices, and BRAMs. Using fewer resources allows architecture designs to target small, low-cost FPGA devices.

2.4 Overview of the Zynq MPSoC

SoC devices typically comprise microprocessors, memories, peripheral interfaces and many other processing components within the same chip. SoC devices have been increasingly used for embedded vision applications as they feature low power consumption and small physical size. Recently, FPGAs have been integrated into SoC devices to leverage their parallel processing capabilities in embedded applications. An example of this is the Zynq MPSoC, which combines a hard silicon Processing System (PS) with FPGA Programmable Logic (PL), fabricated on the same chip [47]. The FPGA logic fabric can accelerate computational algorithms and tasks, while the PS executes an operating system and maintains application control.

This section provides an overview of the AMD Zynq MPSoC and introduces the Python on Zynq Framework, commonly referred to as PYNQ, which is a software framework for Zynq MPSoC devices [25]. The Zynq MPSoC is the target platform for the FPGA architectures detailed in Chapters 4, 5, and 6.

2.4.1 The Zynq MPSoC Architecture

The Zynq MPSoC is an evolution of the Zynq-7000 SoC [48]. In comparison to the Zynq SoC, the Zynq MPSoC contains a significantly improved PS and Kintex UltraScale+ FPGA logic fabric [47]. The PS is composed of an Arm Cortex-A53 Application Processor Unit (APU) [49] and an Arm Cortex-R5 Real-Time Processor Unit (RPU) [50]. The majority of Zynq MPSoC devices feature a Mali-400 GPU [51] and a subset of these devices host a Video Codec Unit (VCU) [52] in the PL. An overview of the Zynq MPSoC architecture is illustrated in Figure 2.6.

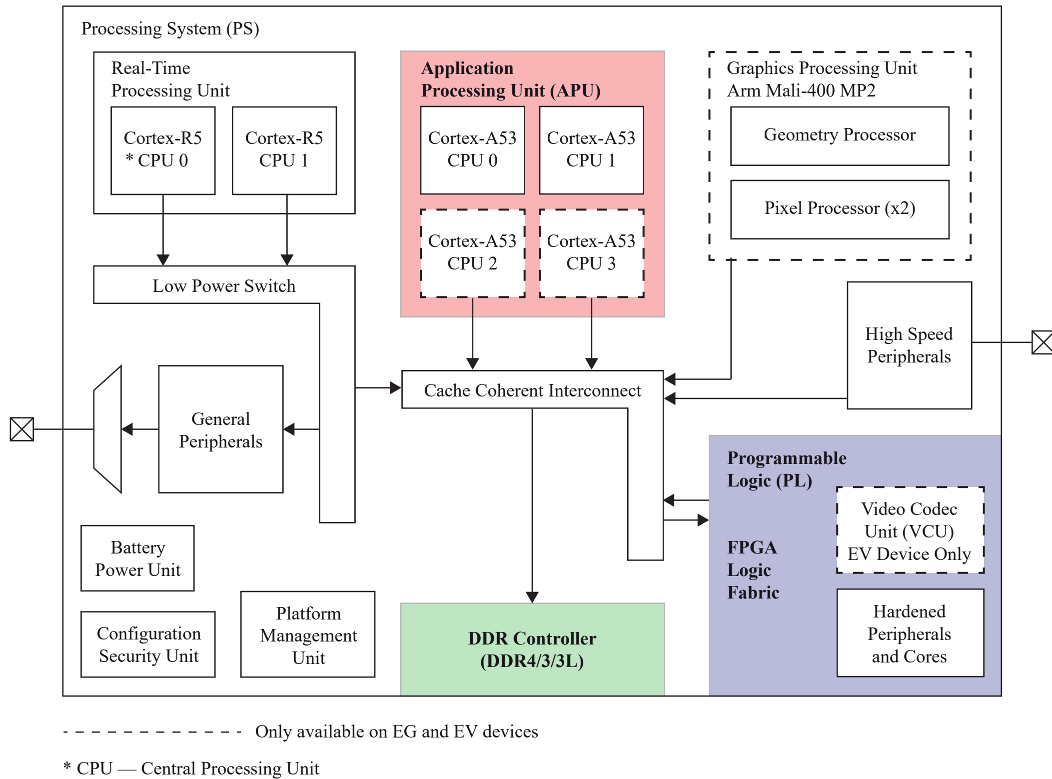


Figure 2.6: An overview of the Zynq MPSoC device architecture.

The FPGA and microprocessors each have unique capabilities and features for applying algorithms and tasks. However, the work contained in this thesis only requires the APU, FPGA, and Double Data Rate (DDR) memory controller components of the Zynq MPSoC. Therefore, discussion will be limited to these components only.

2.4.2 The Advanced eXtensible Interface

The Arm Advanced Microcontroller Bus Architecture (AMBA) open standard [53] is primarily used to transfer data throughout the entire Zynq MPSoC. This standard describes the Advanced eXtensible Interface 4 (AXI4) protocol, which provides processing elements with high-bandwidth, low latency communication to other system components. At the time of writing there are three AXI4 protocols; each are summarised as follows:

- AXI4 [54] — A memory-mapped protocol that transfers data using single beat or burst transactions. Single beat transfers only send one element of data per transaction. Burst transfers send multiple elements of data using one transaction. This protocol is suitable for transferring large quantities of data with up to 256 data beats per burst transfer.
- AXI4-Lite [54] — AXI4-Lite is similar to AXI4, however, only single beat transfers are possible (due to reduced handshaking signals). This protocol is suitable for low-bandwidth communication with hardware control and status registers.
- AXI4-Stream [55] — This protocol is used for point-to-point streaming and is capable of data transfers of infinite size. The AXI4-Stream protocol is suitable for signal processing in vision and radio applications.

To achieve maximum performance, AXI4 should be used as it can achieve burst transfers consisting of multiple data beats. If low-bandwidth communication is required, FPGA resource consumption can be reduced by using the AXI4-Lite protocol instead. Lastly, if it is necessary to use point-to-point streaming, then the AXI4-Stream protocol will be required. A simple example of how these interfaces may be used is detailed in Section 2.4.4.

2.4.3 The PS and PL Interface

The primary attraction of Zynq MPSoC devices for embedded applications is their combination of dedicated silicon processors in the PS, and FPGA logic fabric in the PL. The Zynq MPSoC enables developers to leverage hardware and software co-processing on a single chip. To support communication between the Zynq MPSoC's PS and PL, a set of interfaces are available that allow data to be effectively moved between these processing resources. In this section, a subset of PS-PL interfaces will be described and their naming conventions explained.

The range of interfaces available between the Zynq MPSoC's PS and PL leverage the AMBA open standard for communicating between processing resources, mainly the AXI4 protocol. The work in this thesis only requires two of these interfaces, which are named the high-performance slave port and the high-performance master port. These interfaces are used for different purposes as will be explained shortly. Firstly, each interface adopts a naming convention, as follows:

- High-performance slave — S_AXI_HP[0:3]_FPD.
- High-performance master — M_AXI_HPM[0:1]_FPD.

Each port name contains the acronym AXI, which represents the port's communication protocol. The port names also contain the acronym FPD, which stands for Full-Power Domain. The Zynq MPSoC contains several power domains, which are groups of processing elements that share similar power supply characteristics. The power domains are independent of one another and can be switched off to save power. Including the associated power domain in the port name is useful for the system designer and allows them to target a particular power domain.

The primary reason for using a high-performance slave port is to allow the Zynq MPSoC's PL control AXI data transactions to the PS. Alternatively, the Zynq MPSoC's PS should use the high-performance master port to control AXI data transactions to the PL. Each port is capable of high-performance data transfers, meaning they should be used to transfer large bursts of data. These interfaces will be discussed again in Chapter 4, as they are required to transfer data between the PL and PS.

For completeness, several diagrams throughout this thesis will contain other Zynq MPSoC PS-PL interface names that have not been described here. For further information on the remaining PS-PL interfaces, the reader is directed to [30].

2.4.4 Data Movement and Communication

FPGA hardware accelerators can be used by the APU to improve the computation time of algorithms and tasks. The information required to create a hardware accelerator is commonly packaged into an Intellectual Property (IP) core, which can be designed using the methods and tools outlined in Section 2.5. Hardware accelerators are typically controlled by the APU using an AXI4-Lite interface. When transferring large quantities of data between the FPGA logic fabric and the APU, the full AXI4 memory-mapped protocol is required to reduce the overhead of transfers using burst mode. An example of a typical vision system on the Zynq MPSoC that exploits hardware acceleration is illustrated in Figure 2.7.

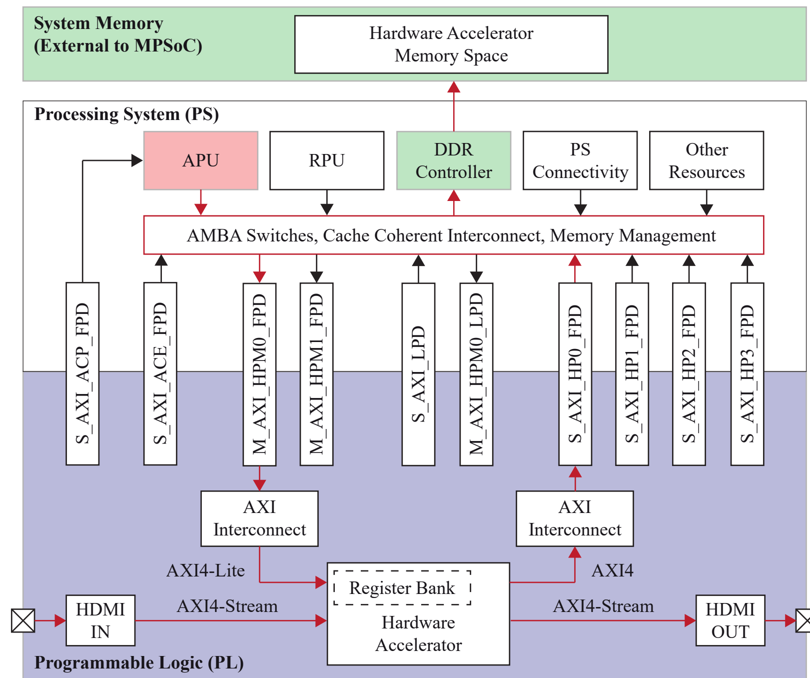


Figure 2.7: An example of a vision system with an HDMI interface and hardware accelerator implemented on the Zynq MPSoC device. The hardware accelerator can communicate with the external memory system and use it to buffer data and store run-time information.

As presented in Figure 2.7, a hardware accelerator is implemented in the FPGA logic fabric and is controlled by the APU using an AXI4-Lite interface. This particular system receives High-Definition Multimedia Interface (HDMI) data using the AXI4-Stream protocol, and is capable of achieving burst data transfers to the DDR memory using the AXI4 memory-mapped interface. Note the use of the high-performance master port and high-performance slave port between the PS and PL of the Zynq MPSoC.

Moving data between the PS and PL enables the development of applications that can exploit a heterogeneous processing environment. The work in this thesis does not explore the possibilities of using a heterogeneous environment to improve application performance. However, from a research and development perspective, a heterogeneous environment provides new possibilities for the evaluation of data and hardware validation of FPGA architectures. For example, the APU can be used to manipulate and evaluate data received from the FPGA, which can simplify the analysis of results or the creation of stimulus for custom FPGA designs. This concept is explored in Chapter 4.

2.5 Design Tools and Development Board

There are various development tools for creating applications that target the Zynq MPSoC. The work in this thesis uses MathWorks *HDL Coder* [26], the Vivado Design Suite [56], and the PYNQ software framework. Figure 2.8 contains a simple workflow diagram demonstrating how to use these tools to develop a Zynq MPSoC application. MathWorks *HDL Coder* is used to develop FPGA architectures and generate IP cores. The Vivado Design Suite is essential for IP core integration. The PYNQ framework supports embedded software development and FPGA architecture validation, which is discussed further in Chapter 4. This section introduces these design tools and details the target platform used throughout this thesis, the ZCU104 development board.

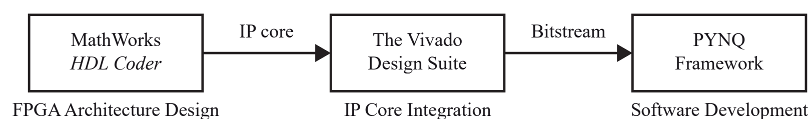


Figure 2.8: A basic workflow diagram illustrating the relationship between the Zynq MPSoC development tools used in this work.

2.5.1 MathWorks *HDL Coder*

FPGA architectures can be designed using a Hardware Description Language (HDL) such as Very High Speed Integrated Circuit HDL (VHDL) or Verilog. These languages provide fine grained control over FPGA architecture design and enable the developer to efficiently target FPGA resources. However, developing systems using HDL can be time consuming. There are many design tools that speed up architecture development time by providing a higher level of design abstraction. One of these tools is MathWorks *HDL Coder*. This tool operates in the MathWorks *Simulink* environment [57]. Simulink enables the simulation of architectures before targeting an FPGA device.

MathWorks *HDL Coder* is able to generate Verilog and VHDL code from a library of HDL Simulink blocks, which represent a high level of design abstraction from the target FPGA resources. For example, there are no DSP48E2 blocks in the *HDL Coder* library. Instead, the designer is expected to use the default product, adder, and relational blocks provided in the *HDL Coder* blockset to achieve the same functionality. A subset of the available HDL compatible blocks are shown to the left of Figure 2.9, while the product block and configuration properties are given on the right.

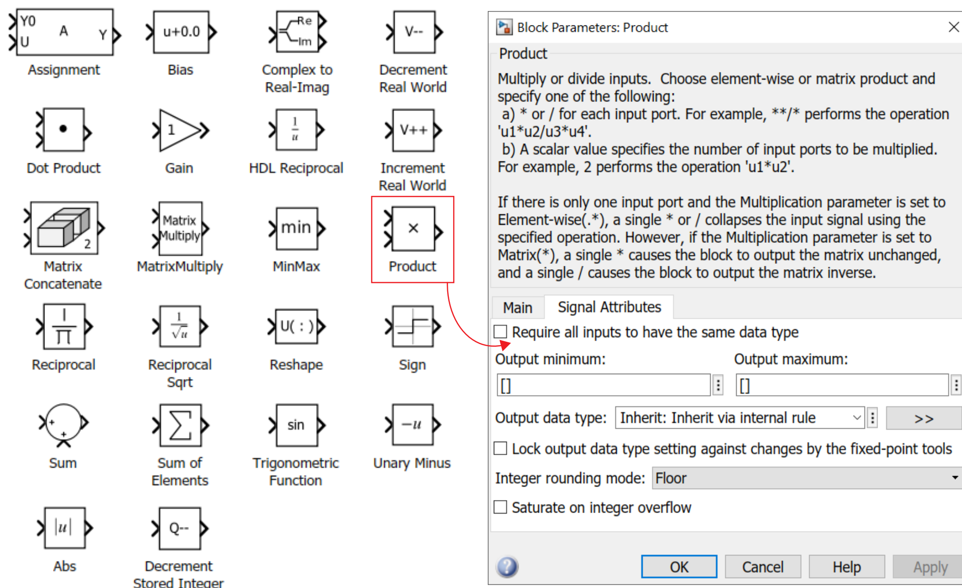


Figure 2.9: A subset of the MathWorks *HDL Coder* blockset (left). The product block and associated parameters are shown (right).

Adopting *HDL Coder* for FPGA architecture design and embedded system integration provides a number of advantages. The output HDL code can be entirely inspected by the user after generation. The designer may choose to develop their architecture using floating-point precision, and then later convert the design to fixed-point data types using the same functional blocks. Software validation through simulation performs well as most Simulink blocks are optimised for simulation purposes. Many HDL-compatible blocks support frame-based processing that can operate on a vector of samples per clock cycle, which improves performance and parallelism. These capabilities are excellent reasons for selecting *HDL Coder* as a primary architecture design tool. However, it is worth keeping in mind that low-level architecture control is minimal, which may decrease architecture efficiency.

HDL Coder can be used to generate HDL for an FPGA architecture design, which can then be converted into an IP core. The IP core is then rapidly integrated into an already existing FPGA system. This development workflow is known as an *HDL Coder* reference design. Reference designs are suitable for rapidly developing FPGA systems and enable repeatability between design iterations. An original contribution of this thesis uses an *HDL Coder* reference design to evaluate and compare LHT architectures. This concept is described further in Chapter 4.

2.5.2 Intellectual Property (IP) Cores

An IP core is a package containing information on how to construct an ASIC or FPGA circuit that performs a specific task. The information may be HDL code or a recipe using IP cores in other repositories. The circuit layout inside the IP core is the intellectual property of an individual or group that owns the legal rights to the design. The IP core can be licensed to other parties to use the circuit in their ASIC or FPGA designs. The Vivado Design Suite has an IP catalogue containing many free-to-use and license-based IP cores.

IP cores are customised using HDL generics during system integration. Figure 2.10 contains an example of an AXI Direct Memory Access (DMA) IP core [58] and its associated configuration properties from the Vivado IP catalogue. The work in this

thesis uses IP cores as a mechanism for creating and integrating Zynq MPSoC designs. As later shown in Chapter 4, an IP core is generated using MathWorks *HDL Coder* for integration into a Zynq MPSoC system to enable rapid prototyping.

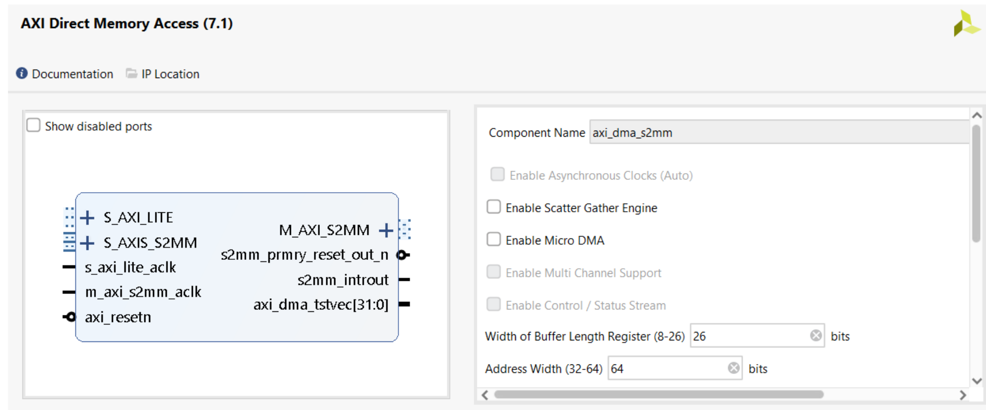


Figure 2.10: An AXI DMA IP core and its associated configuration properties. This image was generated from Vivado's IP catalogue.

2.5.3 The Vivado Design Suite

AMD provides a dedicated suite of tools for developing embedded systems for its FPGAs and Zynq-based devices. These tools include an IP core packager, Software Development Kit (SDK), the System Generator block-based design tool [59], the Vivado High Level Synthesis (HLS) tool [60], an HDL development environment and simulator, and *IP Integrator*.

The IP Integrator tool will be of particular importance for the work in this thesis as it is useful for combining IP cores from different sources into a single monolithic system. An example of interconnecting two IP cores using IP Integrator is shown in Figure 2.11. Notice that the AXI DMA is connected to the LHT architecture using a communication interface, which is AXI4-Stream in this example. The IP Integrator tool also allows users to configure physical FPGA design constraints such as input/output pins and operating standards. Particular design requirements such as FPGA resource selection, resource placement, clock constraints, and signal routing can also be specified. Many design constraints can be applied using the IP Integrator graphical interface, or by using a Xilinx Design Constraints (XDC) file to automate the process.

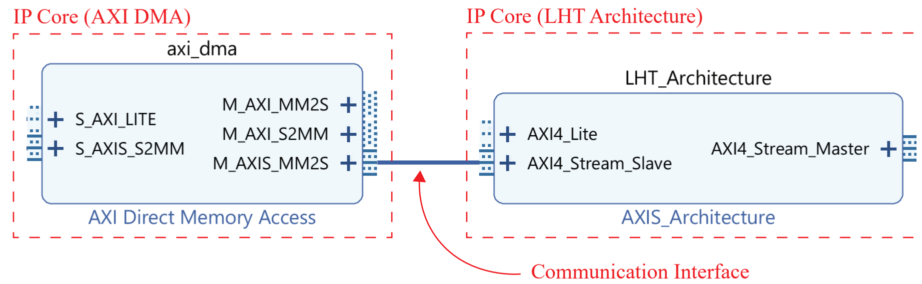


Figure 2.11: An example of interconnecting two IP cores using the IP Integrator tool. The communication interface shown in the diagram uses the AXI4-Stream protocol.

The entire Vivado design suite is integrated with a scripting language known as Tool Command Language (Tcl). Tcl is useful for executing automated scripts, interactively performing design tasks, and querying properties of the associated project. XDC files also use Tcl semantics to simplify the process of implementing design constraints. A particularly useful aspect of Tcl is its ability to automatically generate entire IP Integrator projects and generate output products such as FPGA bitstreams for programming the hardware design onto the chip. In Chapter 4, Tcl scripting is used to automate an IP Integrator project for rapid development of LHT architectures.

2.5.4 PYNQ: The Python Productivity for Zynq Framework

PYNQ is a software framework for Zynq-based embedded systems. The framework uses Python-based programming to simplify the development of applications for SoC platforms, such as the Zynq MPSoC. The PYNQ software stack is composed of four different layers as presented in Figure 2.12. The stack contains a hardware layer consisting of user-defined FPGA bitstreams, an operating system layer that uses a Linux kernel, a Python programming layer to simplify software design, and an applications layer for user interactivity and application development.

The FPGA bitstream layer is a user-defined hardware system design, which is normally referred to as an overlay. Overlays are typically developed for specific types of applications and often provide general facilities that allow them to be reused across similar tasks. These types of hardware designs promote reuse and sharing and often contain many different hardware accelerators per bitstream.

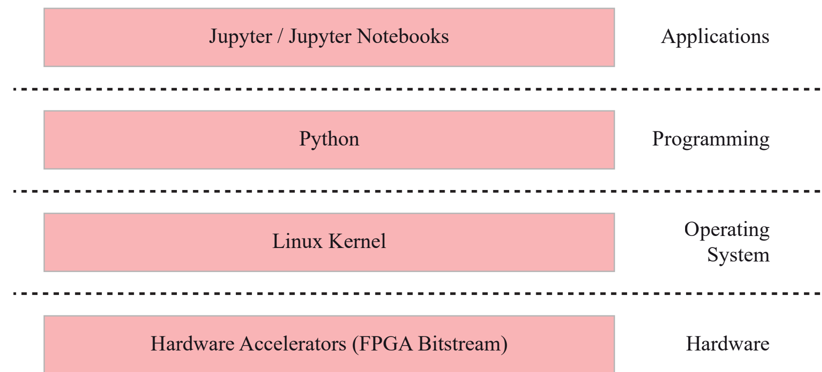


Figure 2.12: The layers in the software stack of the PYNQ framework.

The primary motivation for using PYNQ is to simplify the analysis and evaluation of custom FPGA architectures presented in Chapters 5 and 6. The use of PYNQ as an evaluation tool for FPGA architectures of the LHT is described further in Chapter 4. The remainder of this section will detail the use of the Python programming language and Jupyter (a data science application) in PYNQ.

Python

In summary, Python [61] is a high-level programming language that uses an interpreter instead of compiling instructions into machine code. PYNQ uses Python to speed-up development time and improve code readability (in comparison to compiled languages such as C or C++). Overlays can be developed and associated with custom Python drivers that control the system operation. Drivers abstract the complexity of the underlying hardware accelerators using simple Python code, which can be used by those unfamiliar with hardware design.

Jupyter

Jupyter [62] is an open-source data science project with the primary goal of creating 'reproducible science'. The Jupyter project enables researchers to present and validate their findings by sharing an interactive document known as a Jupyter Notebook. Creating, viewing, and interacting with a Jupyter Notebook can be achieved using a web browser. Figure 2.13 presents an example of a Jupyter Notebook.

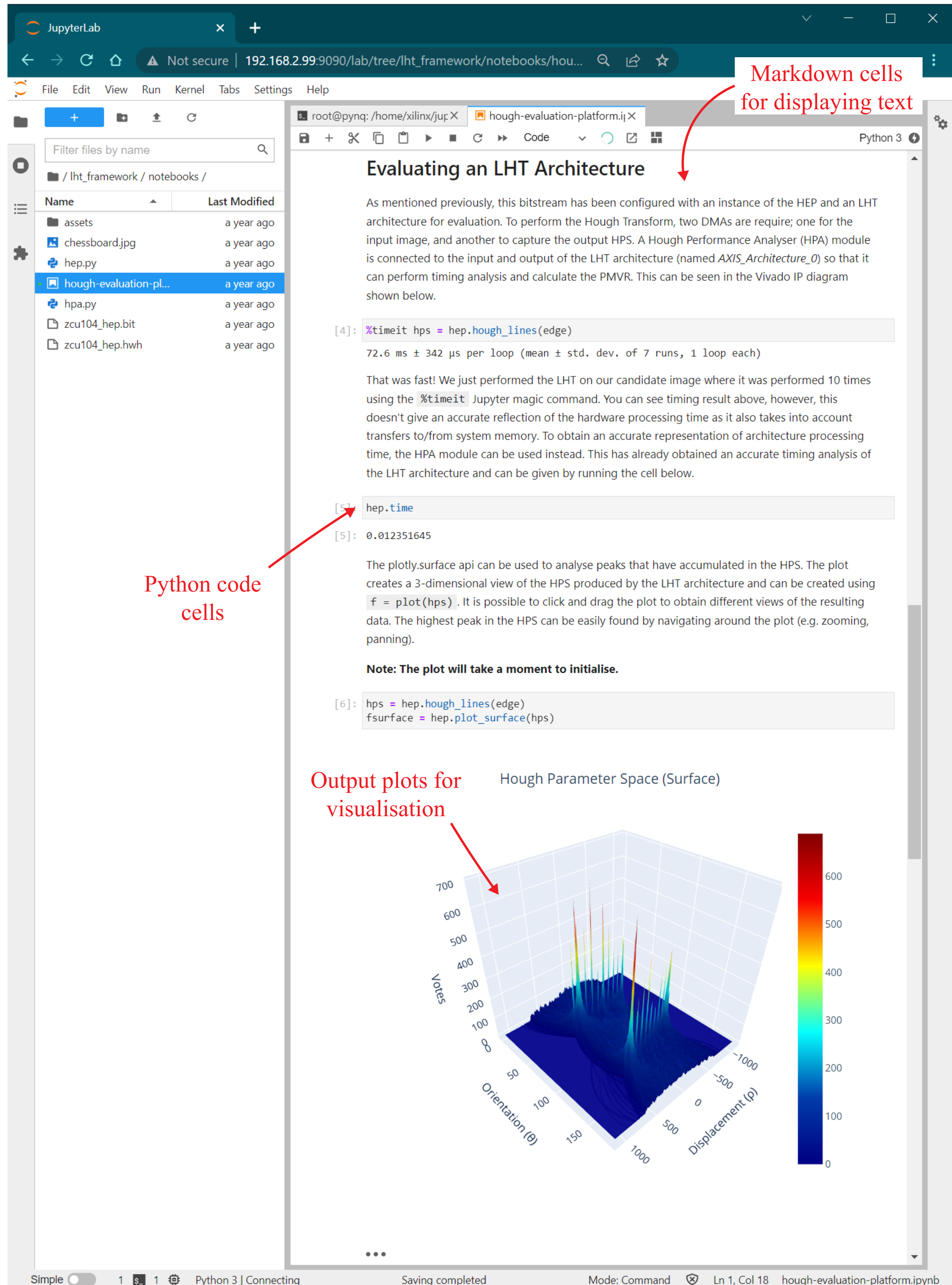


Figure 2.13: An example of a Jupyter Notebook containing rich markdown text cells, executable Python code cells, and output plots for visualisation.

Jupyter Notebooks consist of many different graphical interfaces including interactive plotting, markdown text and Python code cells. PYNQ uses Jupyter Notebooks to enable users to interact with an overlay design. Inspecting and manipulating data transferred between the FPGA and APU is possible and is known as *introspection*. The Jupyter environment allows FPGA developers to confirm the functionality of custom hardware architectures using introspection and also share their results with others.

The RFSoC PYNQ Project

The author of this thesis helped pioneer the AMD RFSoC PYNQ project that also uses Jupyter. The AMD RFSoC [63] is a very similar device to the Zynq MPSoC. However, it is targeted towards radio and instrumentation applications and incorporates high-speed samplers in a single chip. The purpose of the RFSoC PYNQ project was to demonstrate the visualisation and control capabilities of PYNQ for system introspection in a radio communications context. Figure 2.14 presents an example of a visualisation and control interface that displays radio data and allows the user to interact with the RFSoC platform through Jupyter.

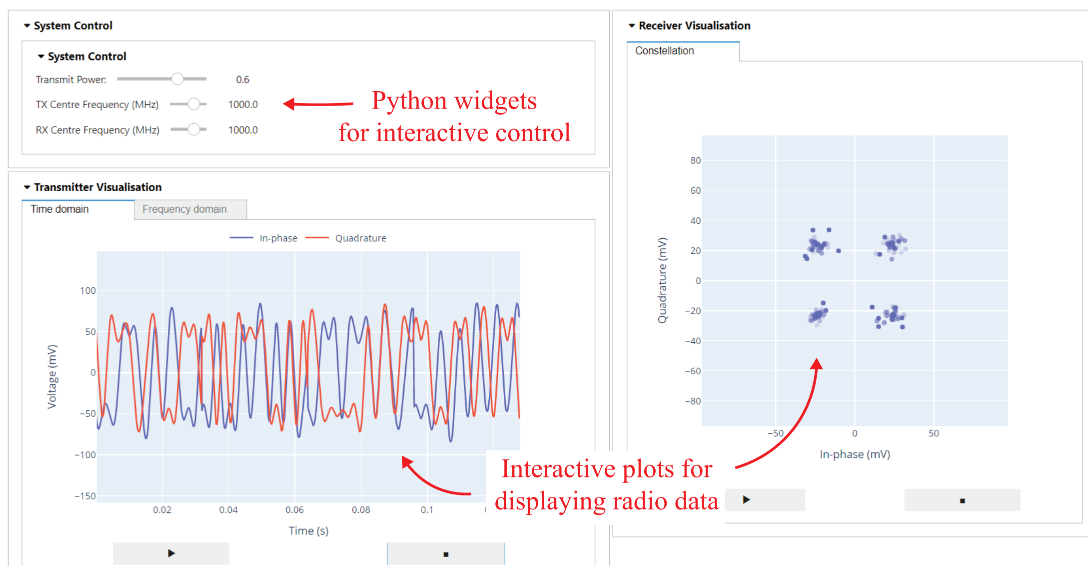


Figure 2.14: An example of a radio introspection application for RFSoC PYNQ.

As shown, the Jupyter environment was used for its plotting and control capabilities. The RFSoC PYNQ project was very successful and published in IEEE Access [29].

2.5.5 The ZCU104 Development Board

The ZCU104 development board [64] offered by AMD is the target platform for implementing LHT architecture designs in this thesis. Figure 2.15 contains a photograph of the ZCU104 development board with labels highlighting the Ethernet interface, power connector, DDR memory, and the Zynq UltraScale+ MPSoC XCZ7UEV-2E device hosted on the board.

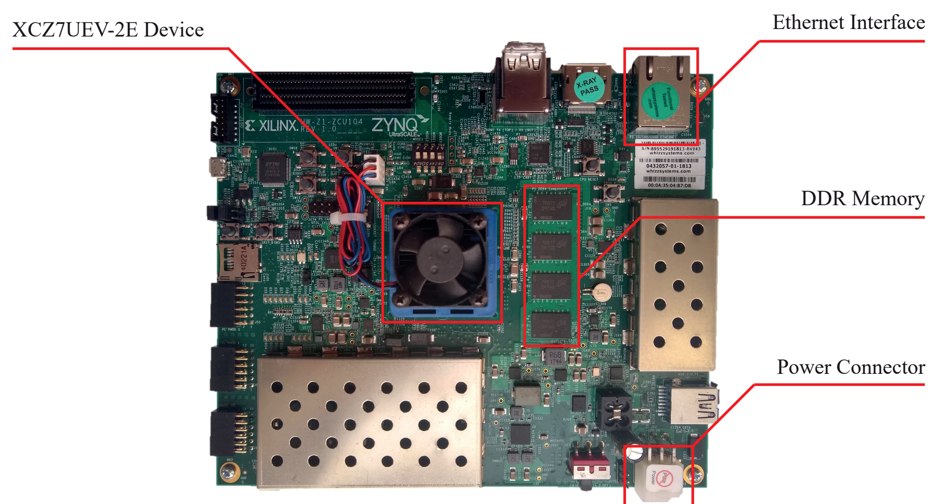


Figure 2.15: The ZCU104 development board that will be used throughout this thesis as the target platform for implementing LHT architecture designs.

The ZCU104 development board is targeted towards computer vision and image processing applications. The XCZ7UEV-2E device hosted on the development board consists of a large FPGA, which contains the resources given in Table 2.3 for implementing architecture designs.

Table 2.3: FPGA resources available on the XCZ7UEV-2E device.

Resource	LUTs	LUT RAM	FFs	BRAM	DSP48E2
Quantity	230,400	101,760	460,800	312	1,728

The novel contributions and LHT architectures presented in Chapters 4, 5, and 6 of this thesis target the ZCU104 development board. The LHT architectures are assessed based on their total consumption of the FPGA resources presented in Table 2.3.

2.6 FPGA & SoC Vision Architectures

Vision systems are often characterised by multiple stages, where each apply an operation or process to an image. In hardware systems, this is known as a vision pipeline and it can be implemented using an FPGA to accelerate computation. Recent interest in designing flexible vision pipelines has prompted the use of heterogeneous processing devices, such as the Zynq MPSoC. Combining dedicated microprocessor units and hardware accelerators is highly effective in achieving dynamic vision designs.

In this section, serial and parallel architectures for applying vision algorithms are described. Furthermore, general pixel-streaming architectures and SoC vision systems are discussed. Finally, spatial filters, edge detection, and binary morphological operations are introduced and their FPGA architectures are presented. These image processing operations and their associated hardware architectures are essential for the work presented in this thesis.

2.6.1 Hardware Accelerated Image Processing

FPGA vision systems typically consist of one or more image processing operations arranged in series. An example is illustrated in Figure 2.16, which contains four stages: preprocessing, segmentation, classification, and recognition. The purpose of preprocessing is to highlight or suppress particular information in an image. Preprocessing could involve image resizing, cropping, sharpening, smoothing, or noise reduction. The next stage, segmentation, groups pixels together that share similar characteristics. Segmentation has the effect of creating image regions, which can be achieved using operations such as thresholding, region-based segmentation, or colour labelling. Classification involves converting image regions into features, which can be described as important primitives for characterising the contents of an image. Finally, the purpose of recognition is to interpret image features and derive a description of an image. The image description that is generated by the recognition stage is typically transferred to a control system that will subsequently execute a procedure or task. Alternatively, the image description can be passed to the system user for analysis.

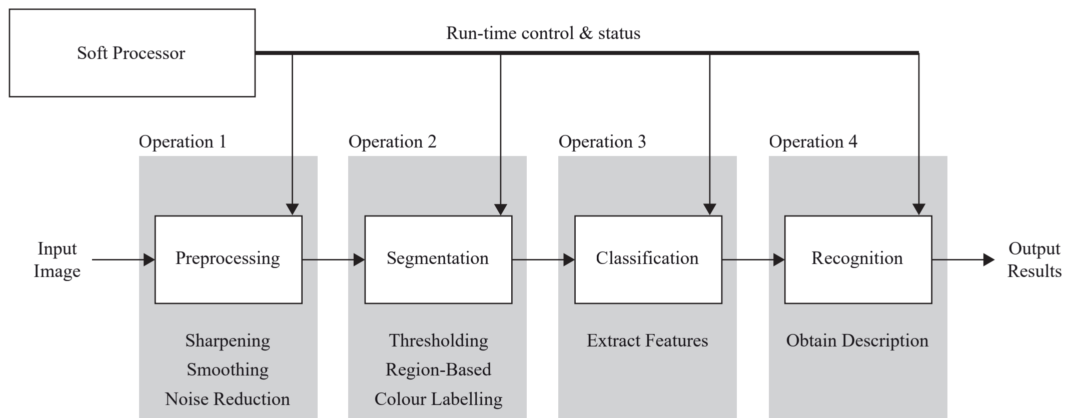


Figure 2.16: An embedded vision system implemented on an FPGA.

Figure 2.16 also contains a soft processor, which is a processor that is implemented on a programmable logic device such as an FPGA. Soft processors enable run-time control of hardware accelerators and provide general purpose processing capabilities that can enable floating point support in many FPGA systems. In AMD FPGA devices, the MicroBlaze soft processor is typically implemented on the logic fabric and offers flexible customisations for the intended application [65]. For example, the total memory allocated for RAM, or the number of floating point units can be customised. However, soft processors consume sizeable areas of the FPGA logic fabric and can only operate at a clock speed that is supported by the native hardware. For example, the maximum clock frequency of a MicroBlaze processor on a Kintex UltraScale+ FPGA is 650 MHz [65]. This is slower than the maximum clock frequency offered by the APU in Zynq MPSoC devices. For example, the APU in the XCZ7UEV-2E device can achieve a clock frequency up to 1.5 GHz, offering better processing speed.

Zynq MPSoC devices offer flexible system architectures for vision applications. Consider the vision system illustrated in Figure 2.17. There are three main points of interest that have been labelled 1, 2, and 3. At point 1, the Zynq MPSoC's PL is applying the same vision pipeline that was given previously in Figure 2.16. However, the image processing operations are interconnected via the AXI4 bus. Arranging the vision pipeline using this topology exposes each hardware accelerator to the entire PS, which includes the APU, RPU and other processing elements. Furthermore, each image pro-

cessing operation can be executed in any order as they are no longer directly connected to one another. At Point 2, the Zynq MPSoC's APU executes an operating system and offers peripheral driver support for Ethernet and USB communication (and many other peripheral interfaces). Finally, at point 3, an Ethernet connection to a larger network is shown. The Zynq MPSoC's APU can host a custom web server to provide an application interface to system users. An application may also leverage the PYNQ software framework to enable Jupyter Notebook capabilities in a web browser.

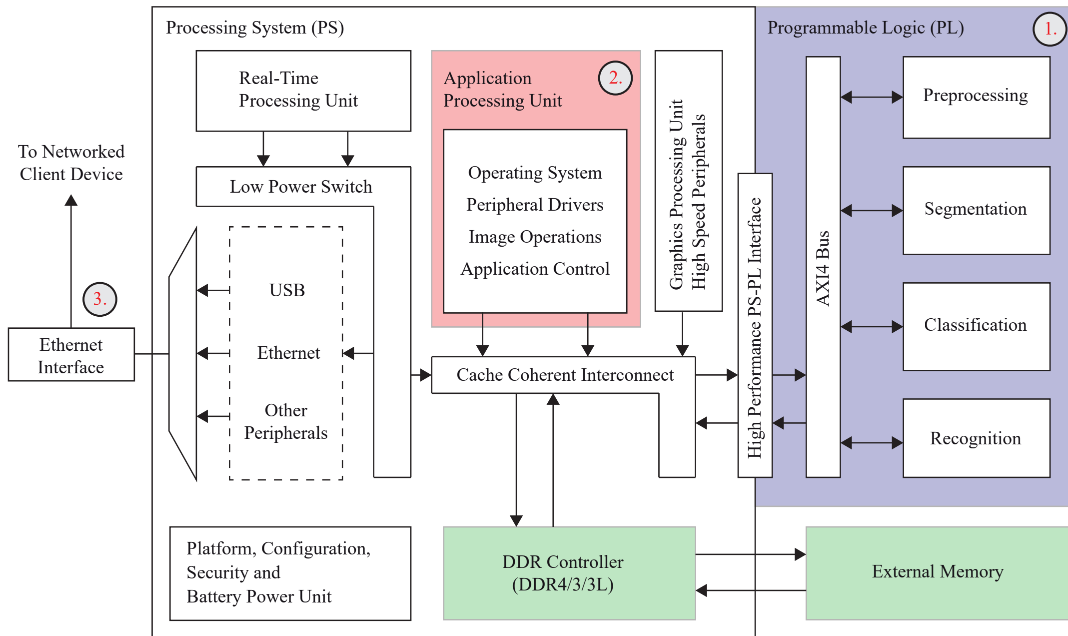


Figure 2.17: An embedded vision system implemented on the Zynq MPSoC device.

Many of the points discussed above can be implemented on a standard FPGA device that uses a soft processor core. However, the primary advantage of using the Zynq MPSoC is to leverage its high-performance, dedicated processor resources to carry out tasks and routines. Instead, soft processors such as the MicroBlaze should be used to support hardware subroutines and functions. Finally, if the PYNQ framework was implemented on a MicroBlaze processor, the performance would be considerably poorer in comparison to a similar implementation using the Zynq MPSoC's dedicated APU.

2.6.2 Stream Processing

Image and video data are often described as 'streaming' through the FPGA logic fabric. Streaming refers to the continuous flow of data between one point and another. For example, an image can be streamed row-by-row between FPGA logic elements (also known as raster scan). As shown in Figure 2.18, this method presents each individual pixel of an image to the input of an arithmetic circuit.

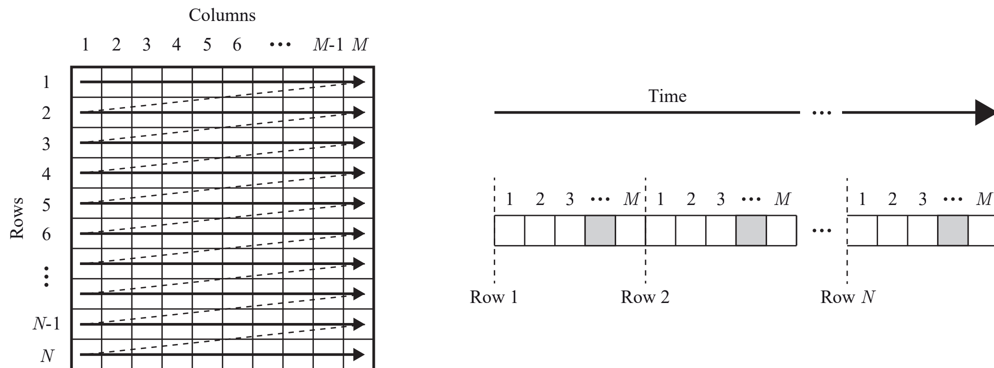


Figure 2.18: Scanning an $M \times N$ image (left), data stream of the original image (right).

Depending on the FPGA architecture, it is possible to stream an image column-by-column, or as an arbitrary group of pixels that are arranged in blocks. However, most FPGA vision systems stream image data by row.

2.6.3 Pixel Intensity

Digital images consist of an array of pixels. An imaging system acquires pixels through a process known as sampling and quantisation, detailed further in [1]. Pixel intensity is a common way of describing the brightness or darkness of a pixel. For instance, high pixel intensity represents greater brightness, while low pixel intensity corresponds to darker tones. Many image digitisation systems have a range of discrete pixel intensity levels. For example, greyscale digital images typically use pixel intensities in the range [0, 255]. The discrete pixel intensity levels are usually positive integers and spaced equally apart.

FPGA architecture designs usually leverage fixed-point data types to store pixel intensity values. These architectures use fewer resources and exhibit lower latency than their floating-point equivalents. However, due to quantisation and storage limitations, the number of intensity levels a pixel can have is usually an integer power of 2. Therefore, the range of pixel intensities is a finite number of levels, L, where $L = 2^k$, and k is the number of bits representing the pixel intensity. For example, we can use $k = 8$ bits for a greyscale image to obtain $L = 256$ intensity levels. It is worth mentioning that colour images also share a similar pixel intensity scheme. However, each colour component has its own intensity value. For instance, a colour image containing red, green, and blue (RGB) components will have three values for each pixel in the image.

2.6.4 Image Representation

This thesis adopts the mathematical notation used by Gonzalez & Woods in [66] to represent an image. In particular, an image is denoted by a two-dimensional function, $f(x, y)$, where the coordinates (x, y) are integers. An image of size $M \times N$ can be expressed as the matrix in (2.1), where each matrix element is a pixel in the image.

$$f(x,y) = \begin{bmatrix} f(1,1) & f(1,2) & \cdots & f(1,N) \\ f(2,1) & f(2,2) & \cdots & f(2,N) \\ \vdots & \vdots & & \vdots \\ f(M,1) & f(M,2) & \cdots & f(M,N) \end{bmatrix} \tag{2.1}$$

Alternatively, a digital image may be represented as a matrix I, which is expressed as

$$I = \begin{bmatrix} i_{1,1} & i_{1,2} & \cdots & i_{1,N} \\ i_{2,1} & i_{2,2} & \cdots & i_{2,N} \\ \vdots & \vdots & & \vdots \\ i_{M,1} & i_{M,2} & \cdots & i_{M,N} \end{bmatrix}. \tag{2.2}$$

The digital image representations given in (2.1) and (2.2) will be used throughout this thesis to mathematically describe image processing operations.

2.6.5 Local Filters

All of the pixels that belong to an image are collectively referred to as the spatial domain. There are many techniques to manipulate the spatial domain. However, local filtering is the most common. Local filtering modifies an image to accentuate or suppress particular information [66]. This technique applies a function to a pixel using a window or local neighbourhood of surrounding pixels. It is possible to achieve image processing operations such as smoothing and sharpening with local filtering. An example demonstrating the effect of these operations on a greyscale digital image is given in Figure 2.19.

(a) A greyscale image of a radio tuner.

(b) A greyscale image of a radio tuner after applying a smoothing local filter.

(c) A greyscale image of a radio tuner after applying a sharpening local filter.

Figure 2.19: A greyscale image of a radio tuner (a) that can be smoothed (b) or sharpened (c) using a local filter. Further information on local filter operations that can smooth or sharpen digital images can be found in [66].

In this section, only linear local filters are considered. The expression for filtering an $M \times N$ image, by an a $m \times n$ local filter, w, is

$$q(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t), \tag{2.3}$$

where q is the output image, and a and b are positive integers. As illustrated in Figure 2.20, local filtering centres a window on a pixel in an image. The shape of the window is usually square, where $m = n$ and m is normally odd. The variables a and b are set to $a = (m-1)/2$ and $b = (n-1)/2$ so that the neighbourhood response of the window is symmetrical around a pixel in the image. However, it is possible to use different window shapes depending on the required response. The window passes over all pixels of the input image, where each new position creates an output value in accordance with (2.3).

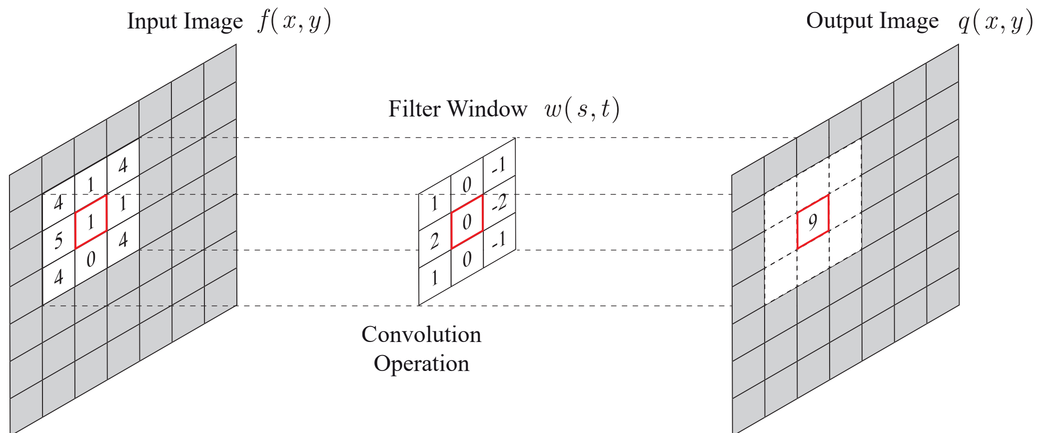


Figure 2.20: A 3×3 spatial filter window $w(s,t)$ centred on a pixel in an image $f(x,y)$.

Convolution is a common operation used to perform linear local filtering by applying a window to an image. To simplify notation, the input image is denoted as a matrix I, which is convolved with a filter window W, to produce the output image Q. This operation is expressed as,

$$Q = W * I. \tag{2.4}$$

Software implementations of local filtering are achieved by sliding the filter window over each pixel of the input image and applying the filter function. Frame buffers are required to store the input and output images during the local filter operation. In contrast, hardware implementations stream the input image through the filter window and avoid using external memories. This technique effectively reduces the complexity of FPGA architecture design and improves the algorithm's latency by avoiding the overhead associated with external memory access.

Local filtering uses each pixel of the input image multiple times in different filter windows. Therefore, caching pixels is essential to ensure that they can be later reused. Additionally, both software and hardware approaches need to consider instances when the filter window is centred on the border of an image. Without effective border management schemes, local filtering can cause artefacts to occur at image borders, which may be undesirable and negatively impact later stages of a vision pipeline.

The remainder of this section describes existing hardware architectures that implement window caches and effective border management schemes for local filtering. The architectures given can be used to support the implementation of fundamental image processing algorithms, such as edge detection.

Window Caches

Window cache architectures require several image row buffers to temporarily store pixel data [67]. Row buffers are useful as they are capable of storing an entire row of pixels in a given image. For filter windows of size $m \times m$, there must be $m-1$ row buffers to effectively cache pixels and allow them to be reused in multiple windows.

A 3×3 local filter architecture that uses row buffers is illustrated in Figure 2.21. This architecture allows an entire image to stream through the filter window. At each clock cycle, the filter window presents a new neighbourhood of pixels that are operated on using the filter function. These pixels are presented to the filter function using taps from the window's shift registers. The expression given in (2.3) is applied in the filter function to create a new output pixel value.

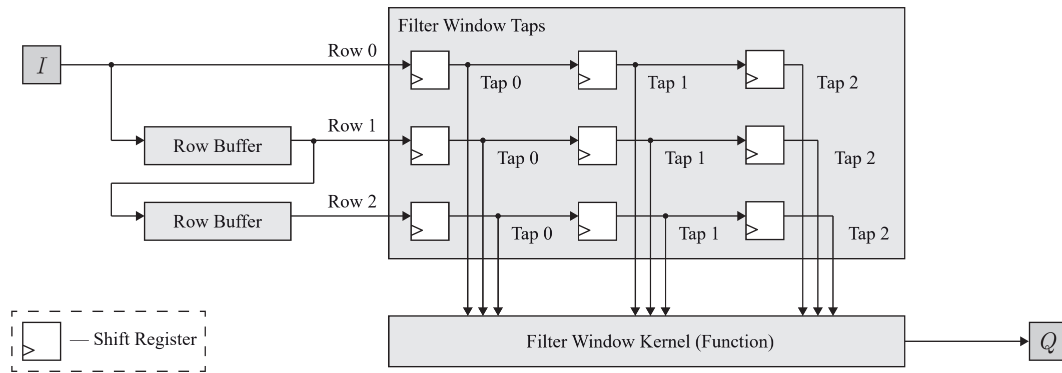


Figure 2.21: An architecture for implementing a spatial filter on an FPGA. This technique arranges row buffers in parallel with the filter window.

Border Management

Effective border management is very important when applying local filters, as it is otherwise possible to create undesirable artefacts at the borders of an image. For example, if we consider an application that detects the number of lines in an image, it is possible that preprocessing operations that use local filtering can unintentionally create additional lines, resulting in errors. One method of mitigating the affects of image borders while local filtering is to only produce output pixels where the filter window fits directly inside the image. Consequently, this technique produces a smaller output image than that of the input. A more effective technique is to initially increase the size of the input image before local filtering. This technique is known as padding and can be performed using a range of different schemes including nearest neighbour extrapolation, constant extension, symmetric padding, and periodic extension [15].

Only nearest neighbour extrapolation will be described, as this technique is used in the architectures for edge detection and binary morphology, which are required later in Chapter 6. Figure 2.22 provides an example of a filter window that can apply a nearest neighbour extrapolation border management scheme. As shown, the filter window has been modified using a set of multiplexers, which redirect the flow of data based on the current image position (or current pixel coordinates). This architecture is a modified version of the work published in [15], in which various techniques are presented that mitigate the effects of local filters at image borders.

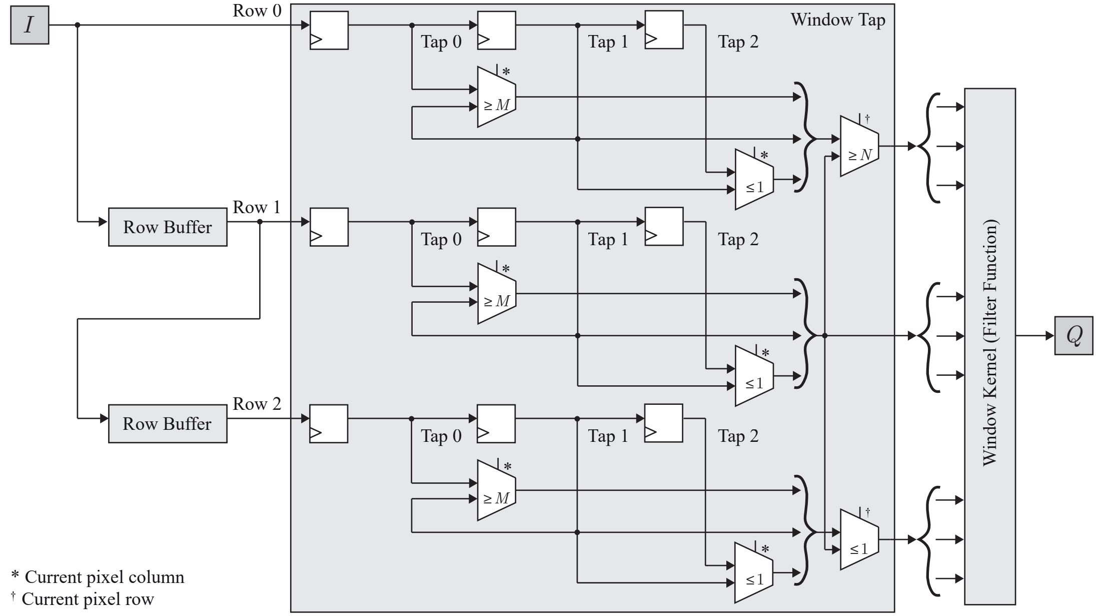


Figure 2.22: An architecture for nearest neighbour extrapolation border management in FPGA implementations of local filters.

All of the border management schemes described in [15] use 5×5 filter windows. The modified architecture presented in Figure 2.22 can apply 3×3 filter windows using the nearest neighbour extrapolation border management scheme. This window size is required to implement edge detection, which is described further in the next section.

2.6.6 Edge Detection

Edge detection is the process of extracting the boundaries of objects within a digital image. This process involves computing the image gradient, which is defined as the directional changes in pixel intensity across the spatial domain [66]. The image gradient is calculated by obtaining the partial derivatives of an image, $f(x, y)$, with respect to its gradient in the x and y direction. These are known as the directional gradients, H and V, which are defined as $\partial f/\partial x$ and $\partial f/\partial y$, respectively. The image gradient, ∇f, is a vector of its partial derivatives, given as

$$\nabla f = \begin{bmatrix} H \\ V \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix}. \tag{2.5}$$

The gradient response of an image can be used to determine the border between two or more image regions. Areas of an image that demonstrate strong gradient intensity are known as edges. To obtain the edges of an image, it is necessary to first calculate the directional gradients. These can be obtained using two linear spatial filters to derive the horizontal and vertical partial derivatives, as given by Prewitt [68] and Sobel [69]. The Canny edge detection algorithm [70] is also very effective. However, its FPGA architecture design is challenging as it is time-consuming to develop and requires a deep understanding of FPGA design techniques. Sobel edge detection will be described, as this is the only technique used to derive an edge image in this thesis.

The Sobel operator convolves two 3×3 filters with an image to derive approximations for the directional gradients, H and V. If we denote the input image as a two dimensional matrix, I, then the directional gradients are calculated as follows

$$H = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \qquad V = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * I \tag{2.6}$$

Upon obtaining the directional gradients, it is possible to derive the magnitude of the image gradient. The gradient magnitude, G, is calculated using

$$G = \sqrt{H^2 + V^2}. \tag{2.7}$$

Calculating (2.7) can be computationally demanding for FPGAs, as it is necessary to apply a square root operation and two multiplications. Alternatively, an approximation of the gradient magnitude is often obtained by taking the absolute value of each directional gradient and summing the results together [71]. This operation is expressed mathematically as

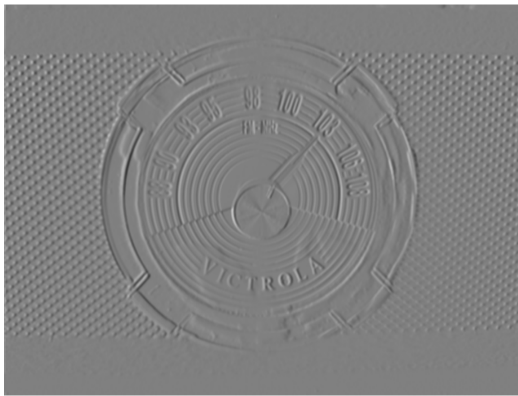
$$G \approx |H| + |V|. \tag{2.8}$$

The edge response can be acquired by segmenting the gradient magnitude using a technique called thresholding, where each pixel is either labelled as the image background or an edge. Thresholding is applied globally to the spatial domain and has the

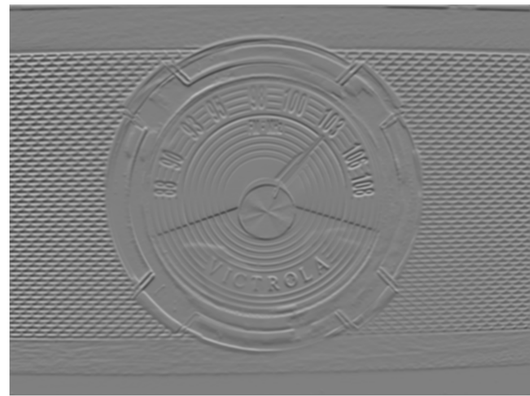
effect of assigning output pixels as true or false (1 or 0) based on whether they are higher or lower than a threshold value, T. The mathematical expression for thresholding the gradient magnitude to produce an edge image, E, is given as

$$E = \begin{cases} 1, & G \geq T \\ 0, & G < T \end{cases}. \tag{2.9}$$

Figure 2.23 presents an example of applying the Sobel operators to extract the edges of the radio tuner image (Figure 2.19a). The example contains four images: the H and V directional gradients, the gradient magnitude, and the edge image.

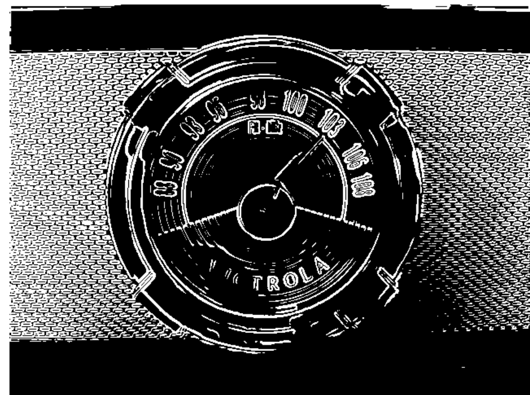


(a) The H directional gradient of the radio tuner.



(b) The V directional gradient of the radio tuner.

(c) The gradient magnitude of the radio tuner.



(d) The edge response of the radio tuner image using a threshold of $T = 160$.

Figure 2.23: The H directional gradient (a) and V directional gradient (b) are used in (2.7) to obtain the gradient magnitude (c) of the radio tuner image. The corresponding edge image (d) is obtained by thresholding the gradient magnitude.

An efficient FPGA architecture that applies the Sobel operators to a greyscale image can be created by decomposing each Sobel filter into two separable filters [67]. The expressions used to derive H and V in (2.6), can be rewritten as

$$H = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \Big(\begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * I \Big) \qquad V = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \Big(\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * I \Big). \tag{2.10}$$

Using two separable filters instead of a two dimensional filter reduces the total FPGA resource consumption. The separable filter approach consumes six adders, while the two dimensional filter method requires twelve adders. Furthermore, there is also a reduction in the number of registers used by the filter window. Only three registers are required in the separable filter approach, while a two dimensional filter consumes nine registers. An FPGA architecture that uses separable filters to apply the Sobel operators to a greyscale image is illustrated in Figure 2.24.

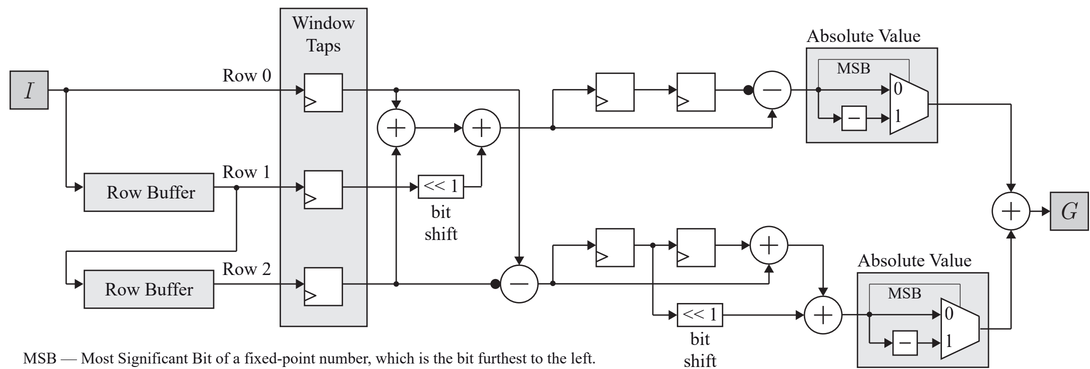


Figure 2.24: An FPGA architecture that applies the Sobel operators to a greyscale image. The architecture design uses separable filters and an efficient gradient magnitude calculation to minimise resource consumption [67].

As shown, the Sobel architecture uses the approximation technique presented in (2.8) to reduce the complexity of calculating the gradient magnitude. Edge pixels are obtained by thresholding the gradient image, which is achieved using a relational operator that selects the output of a multiplexer in accordance with (2.9).

2.6.7 Gradient Orientation

The gradient orientation describes the direction of the largest possible intensity increase within a neighbourhood of pixels. Many algorithms leverage the gradient orientation to obtain the approximate direction of edge pixels in an image. The gradient orientation, α, can be derived using the directional gradients [66], as

$$\alpha = \tan^{-1}\left(\frac{V}{H}\right). \tag{2.11}$$

The COordinate Rotation DIgital Computer (CORDIC) technique [72] can be used to compute the gradient orientation in an FPGA architecture design. CORDIC is an iterative algorithm with several variations that can efficiently compute trigonometric, hyperbolic, and linear functions [73]. Trigonometric CORDIC, also known as Circular CORDIC, is required as (2.11) uses the arctangent of the directional gradients to calculate α. An unrolled architecture of Circular CORDIC operating in vectoring mode is illustrated in Figure 2.25. This architecture can achieve one output per clock cycle, can be pipelined to achieve high clock frequencies, and is suitable for the high throughput processing typically required by video systems.

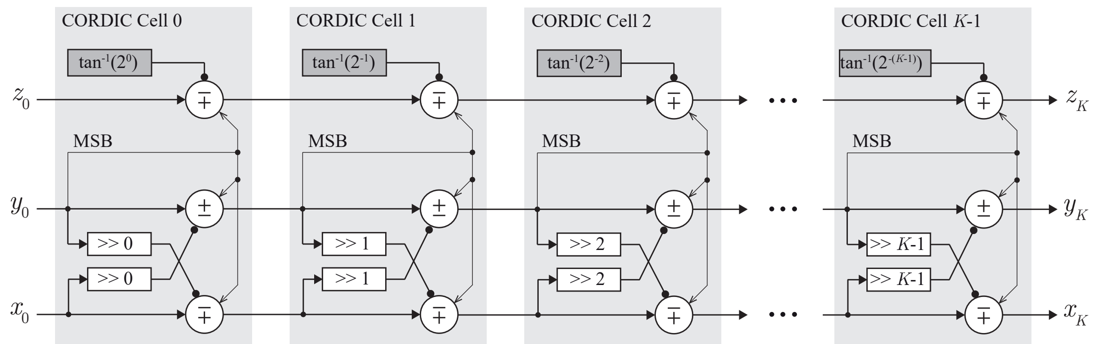


Figure 2.25: FPGA architecture of unrolled CORDIC [74].

The reader is directed to Appendix A for the derivation of the circular CORDIC equations and definition of the input and output variables, which are z_0, y_0, x_0 and z_k, y_k, x_k, respectively. This unrolled CORDIC architecture will be used later in Chapter 6 to derive the gradient magnitude and orientation of a greyscale image.

2.6.8 Binary Morphology

Morphological image processing [66] is based on set theory. It is useful for the analysis and manipulation of geometric structures within an image. Many image processing techniques such as thinning, shape identification, and pruning can be achieved by applying one or more fundamental morphological operations to an image. Morphological operations are achieved by combining an input binary image with a structuring element using a set operator, such as an intersection or inclusion. A structuring element is a shape that is deliberately chosen to emphasise or suppress information in an image.

Morphological operations can be easily applied to a binary image, as each pixel can be designated as the image background or an object. The image background is commonly represented as a binary 0, while an object is represented by a binary 1. The input binary set, X, contains the coordinates of all object pixels in a binary image, I. X can be expressed using set notation as

$$X = \{(x, y) \mid I[x, y] = 1\}. \tag{2.12}$$

The structuring element, B, is a small set of pixel coordinates describing an object used to probe a binary image. Similar to local filtering, the structuring element commonly has odd dimensions so that it is symmetrical when centred on a pixel. Figure 2.26 contains an example of a binary set X and structuring element B. The binary set and structuring element have been converted to a two-dimensional array, where the white and shaded cells represent the object and background, respectively.

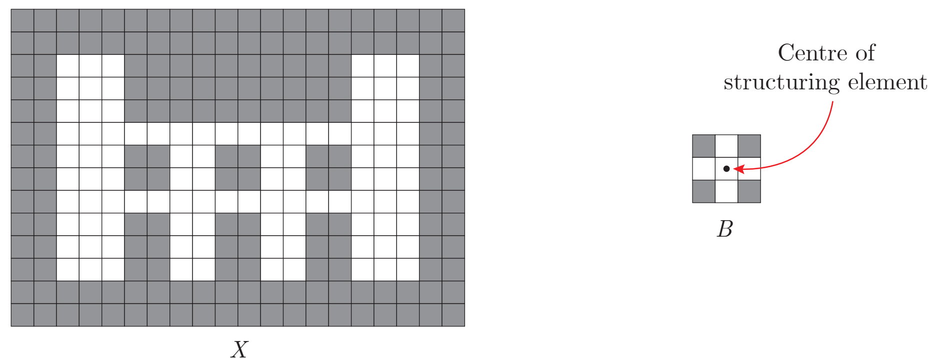


Figure 2.26: An example of a binary set X (left) and structuring element B (right).

This section introduces four binary morphological operations: erosion, dilation, opening, and closing. An example is given for each operation using the binary set and structuring element given in Figure 2.26. This section also describes a corresponding FPGA architecture for each operation. Note that the mathematical notation for morphology adopted by this thesis is similar to that presented in Gonzalez & Woods [66].

Erosion and Dilation

Both erosion and dilation are fundamental morphological operations. They use a structuring element, which interacts with an input binary set, to create a new set of data. Erosion and dilation are often considered opposites, where applying erosion results in fewer object pixels, and applying dilation results in more object pixels.

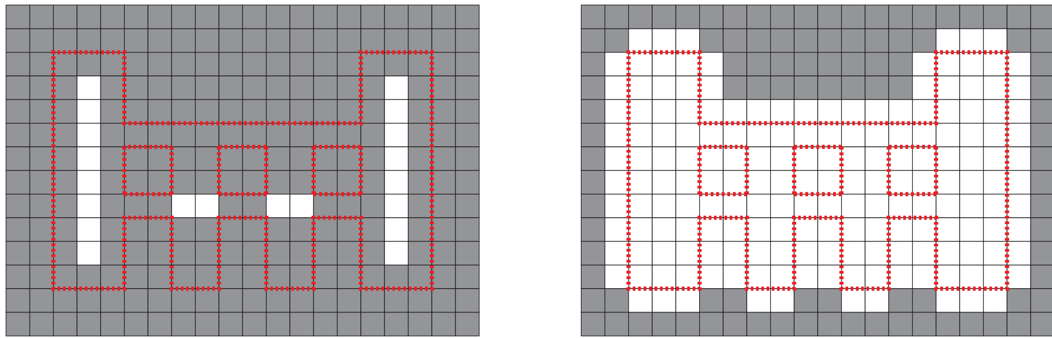
The erosion of a binary set X, by a structuring element B, is a set of all elements in z such that B, translated by z, is a subset of X. Where $z \in \mathbb{Z}$, defined by a set of points (z_i, z_j), which replace the coordinates in B from (x, y) to $(x + z_i, y + z_j)$. A coordinate is produced in the output set when all elements of B are contained in X, $(B)_z \subseteq X$. The erosion of X by B is denoted as $X \ominus B$, and is expressed as

$$X \ominus B = \{z \mid (B)_z \subseteq X\}. \tag{2.13}$$

Dilation operates on the reflection of B about the origin, denoted as $\hat{B}$. The dilation of a binary set X, by a structuring element B, is a set of all elements in z such that $\hat{B}$, translated by z, intersects with X. A coordinate is produced in the output set when the intersection does not result in an empty set, $(\hat{B})_z \cap X \neq \varnothing$. The dilation of X by B is denoted as $X \oplus B$, and can be expressed using

$$X \oplus B = \{z \mid (\hat{B})_z \cap X \neq \varnothing\}. \tag{2.14}$$

Figure 2.27 presents an example of the erosion and dilation of the binary set X by the structuring element B. In the examples presented, X and B are the binary set and structuring element given previously in Figure 2.26. Notice that there are fewer object pixels after an erosion operation and more object pixels after a dilation operation.



(a) The erosion of the binary set X by the structuring element B.

(b) The dilation of the binary set X by the structuring element B.

Figure 2.27: The examples above present the erosion (a) and dilation (b) of the binary set X by the structuring element B. The dashed red outline shows the borders of the original object pixels to help visualise the effects of erosion and dilation on the input binary set.

Erosion and dilation can be implemented as a local filter, where the filter window is the size and shape of the structuring element [67]. The filter function applies a logic operation to all taps in the filter window. These are an AND operation for erosion, and an OR operation for dilation. Both operations can be implemented using a single FPGA architecture as presented in Figure 2.28.

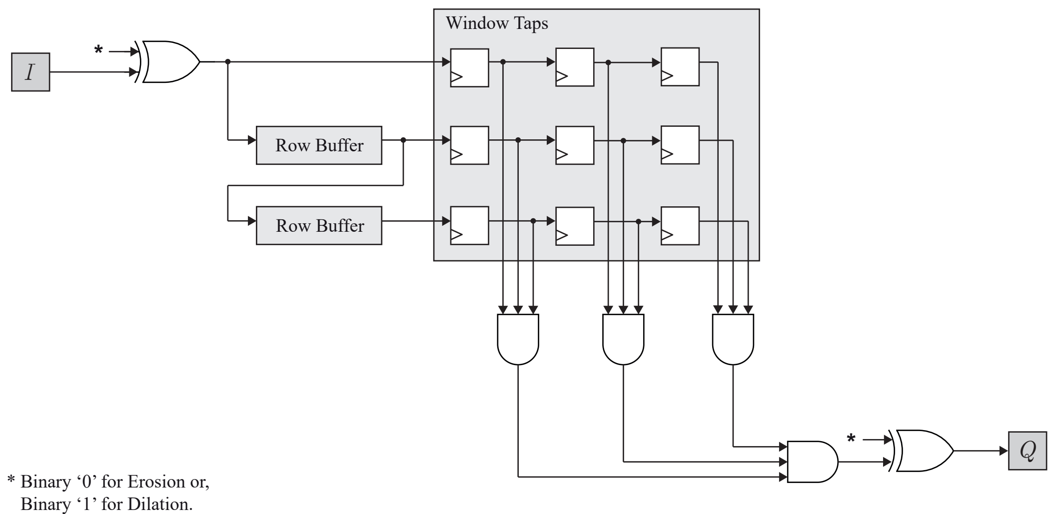
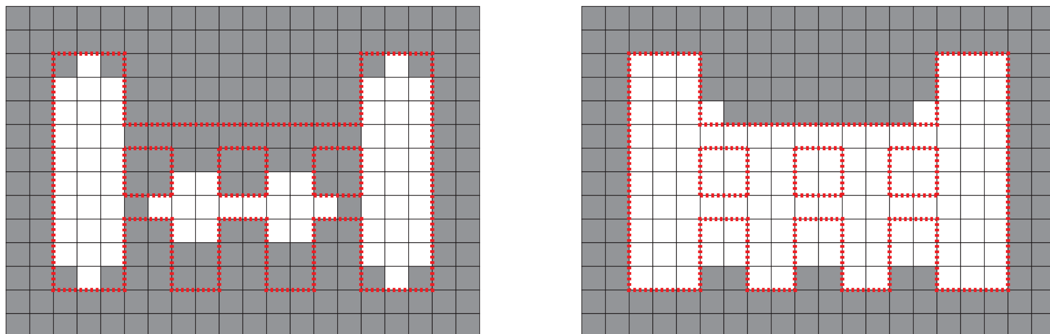


Figure 2.28: An FPGA architecture that applies dilation or erosion to an image using a 3×3 structuring element.

Opening and Closing

Two very common morphological operations are an opening and closing [66]. Opening operations are created by cascading an erosion and dilation. In contrast, closing operations are created by cascading a dilation and erosion. Figure 2.29 presents an example of the opening and closing of the binary set X by the structuring element B. Notice that the opening operation suppresses narrow strips of pixels in the original binary set, while the closing operation removes small holes.



(a) The opening of the binary set X by the structuring element B.

(b) The closing of the binary set X by the structuring element B.

Figure 2.29: The examples above present the opening (a) and closing (b) of the binary set X by the structuring element B. The dashed red outline shows the borders of the original object pixels to help visualise the effects of applying the operations on the input binary set.

Opening and closing operations can be described using set notation. The opening of a binary set X, by a structuring element B, is denoted as $X \circ B$. The erosion operation suppresses features that are smaller than the structuring element, while the dilation attempts to recover the leftover features back to their original size. A binary opening is useful for suppressing 'salt and pepper' noise and removing narrow strips of pixels that connect two or more shapes together. The opening of X, by B, is defined as

$$X \circ B = (X \ominus B) \oplus B. \tag{2.15}$$

The closing of a binary set X, by a structuring element B, is denoted as $X \bullet B$. The dilation combines or enhances connections between shapes, while the erosion suppresses holes that are smaller than the structuring element. A binary closing is useful for filling

in gaps between shapes, and removing small holes in a binary set. The closing of X, by B, is expressed using

$$X \bullet B = (X \oplus B) \oplus B. \tag{2.16}$$

An FPGA architecture that applies a binary opening or closing can be created using the architecture for erosion and dilation presented in Figure 2.28. To implement an opening, the erosion architecture is applied first and the dilation architecture is applied second [67]. The sequence of filters are simply reversed to implement a closing. Note that each morphological operation must use a border management scheme as detailed in Section 2.6.5. Border management is necessary to prevent artefacts occurring at the image borders. The morphological architectures presented in this thesis use nearest neighbour extrapolation border management. In Chapter 6, an FPGA architecture of a morphological opening is used to suppress noise in a two-dimensional binary array.

2.7 Summary

This chapter initially reviewed candidate hardware platforms for implementing embedded computer vision systems. It was found that FPGAs are suitable platforms for accelerating image processing operations as they offer high computational performance and low latency processing compared to other technologies. Kintex UltraScale+ logic fabric and its specialised resources were then explored. The Zynq MPSoC was also introduced, and its constituent components, the PS and PL, were summarised. The AXI4 protocol and data movement between the PS and PL was described. Design tools relevant to the work in this thesis were presented, including the Vivado Design Suite, MathWorks HDL Coder, and the PYNQ software framework. Their advantages towards system prototyping and development were discussed.

The second half of this chapter explored several image processing techniques and their corresponding FPGA architectures, which are relevant to the work in this thesis. Linear spatial filtering and the nearest neighbour extrapolation border management scheme were initially described. The image gradient magnitude and orientation were discussed, and edge detection using the Sobel operators was reviewed. Lastly, morpho-

logical operations for binary images were introduced, and their FPGA implementations were described. The above image processing techniques and corresponding FPGA architectures are essential for the work undertaken in Chapters 4, 5, and 6. The next chapter introduces the LHT for extracting lines in digital images and presents a literature review of previously published works and applications.

Chapter 3

The Hough Transform

3.1 Introduction

The Hough Transform (HT) is a robust technique for extracting objects in digital images. Initially proposed for the extraction of lines [75], the HT can be adapted to extract any analytical and arbitrary object that can be described mathematically [3,76]. The principles of applying the HT to an image are straightforward.

A mathematical function that accurately defines an object in the spatial image domain is chosen to be the basis of the HT kernel. There are many functions (also known as templates) for an object, where the performance of the HT can vary depending on the selected function. An area in memory is preallocated to store a multidimensional array, commonly referred to as the Hough Parameter Space (HPS). Each dimension of the HPS relates to a parameter that is used to describe the object. Initially, the value stored in each location of the HPS is set to zero.

A binary image that contains an object is created using edge detection (Section 2.6.6), or another form of image segmentation. Each edge pixel is processed using the HT kernel to create parameters, as described by the object template. The parameters are used to address a location in the HPS and increment that location by one. This procedure is known as voting, where votes are effectively accumulated until the last edge pixel is processed. The effect of voting produces peaks in the HPS, which correspond to parameters of the object in the edge image.

Due to the voting process, the HT is robust to edge images that contain partially occluded objects. Generally, objects that are missing edges can still be extracted provided there are enough edges present to create a peak in the HPS. The HT is also an effective tool for extracting objects from images containing noise. Edge pixels that are not part of the object (i.e. noise) receive very few votes in the HPS and tend not to interfere with the larger peaks.

The general HT process is illustrated in Figure 3.1. For practical implementation purposes, the HPS is divided into discrete cells that are maintained in memory. When parameters of edge pixels are derived using the HT kernel, they are rounded to the nearest location in the HPS before applying a vote.

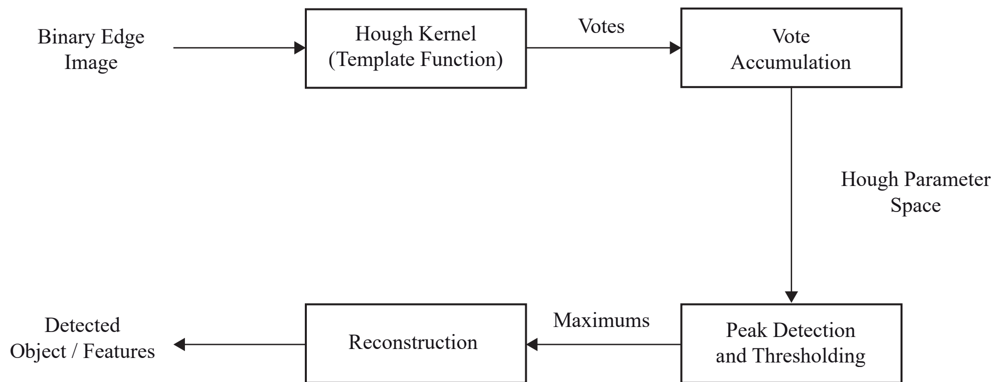


Figure 3.1: The general HT process for extracting an object in a binary image.

The HT is usually named the Line Hough Transform (LHT) when configured to detect lines in a digital image. The work in this thesis aims explicitly to implement the LHT on FPGA and SoC devices and reduce the overall memory requirements of the HPS. This chapter will describe the operation of the LHT and summarise relevant literature consisting of optimisations, variations, and associated hardware architectures. The algebraic notation adopted by Gonzalez & Woods [66] to describe the LHT is used throughout this chapter. Also, to evaluate the memory consumption of the HPS, an expression that calculates the number of memory bits necessary to store the HPS is derived. This expression is required later when comparing the memory consumption of LHT architectures presented in Chapters 5 and 6.

3.2 Line Detection

Rosenfeld was the first to algebraically describe the LHT using the slope-intercept representation of a straight line [77, 78]. The equation for a line $y_i = ax_i + b$ can be used as a template, where (x_i, y_i) is a point in the spatial domain, a is the slope, and b is the y-axis intercept. An infinite number of lines pass through the point (x_i, y_i), where each line has its own parameters (a, b) that satisfy the relationship,

$$b = y_i - ax_i. \tag{3.1}$$

Lines are formed in the spatial domain by two or more pixels that share the same parameters (a, b). Consider a second point (x_j, y_j) that lies on the same line as (x_i, y_i) in the spatial domain. Both points can be used in (3.1) to produce values of b for all values of a in the range $[-\infty, \infty]$. The slope and y-axis intercept of the line containing both (x_i, y_i) and (x_j, y_j) can be determined by plotting the resulting parameters (a, b) in an ab-plane. This operation is demonstrated in Figure 3.2 for points (x_i, y_i) and (x_j, y_j), where the resulting parameters have intersected at point (a', b').

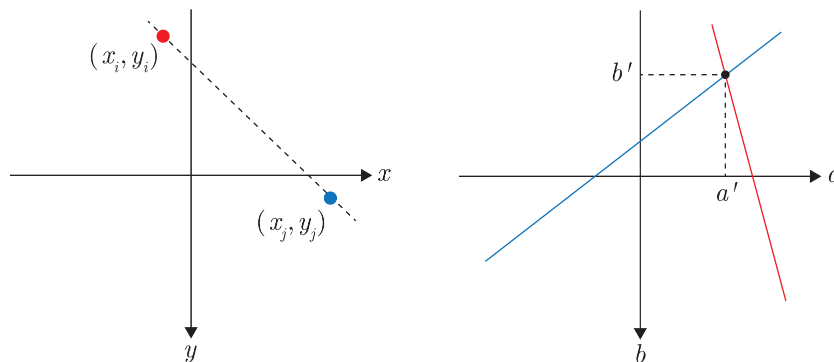


Figure 3.2: The slope-intercept representation of a straight line is used to map points, (x_i, y_i) and (x_j, y_j), in the spatial domain (left) to the ab-plane (right). The points are mapped using (3.1) over a set of real values, a.

The point of intersection (a', b') is the slope and y-axis intercept of the line that contains the points (x_i, y_i) and (x_j, y_j). The slope-intercept representation of a straight line is suitable as a template when extracting lines that are horizontal or approximately horizontal. Lines that are vertical or approximately vertical have large values of slope

and y-axis intercept. For instance, a vertical line causes a to approach infinity. The a-axis of the ab-plane will be very challenging to implement practically, as it would require infinite memory. The solution to this problem is to represent a straight line using an analytical function that relates collinear points to a set of stable parameters.

The normal representation of a straight line was proposed as a template by Duda & Hart [3] where the magnitude of displacement ρ, and the orientation of displacement θ, are used to describe the location of a straight line from a predefined origin in the spatial domain. This relationship is expressed as

$$\rho(\theta) = x_i \cos(\theta) + y_i \sin(\theta), \tag{3.2}$$

where θ is a set of real values in the range $[0, 360°)$, since (3.2) is periodic across intervals of 360°, such that $\rho(\theta) = \rho(\theta + 360°)$. Figure 3.3 demonstrates the mapping of two points in the spatial domain, (x_i, y_i) and (x_j, y_j), to the $\rho\theta$-plane. As shown, voting has the effect of creating sinusoids in the $\rho\theta$-plane, which intersect at (ρ', θ') and $(-\rho', \theta' + 180°)$.

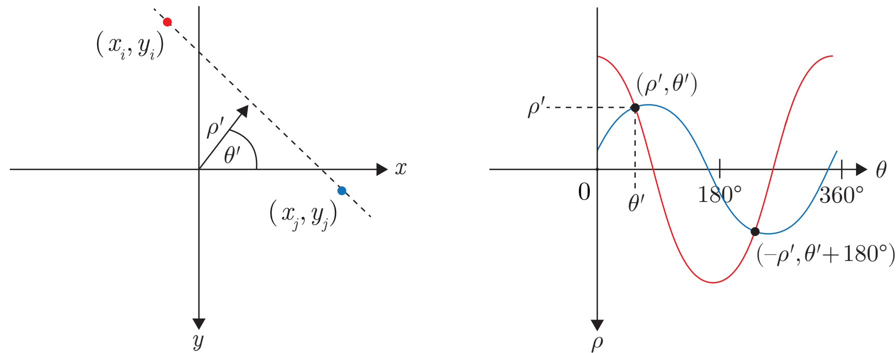


Figure 3.3: The normal representation of a straight line is used to map points, (x_i, y_i) and (x_j, y_j), in the spatial domain (left) to the $\rho\theta$-plane (right). The points are mapped using (3.2) over a set of real values, θ.

The line containing the points (x_i, y_i) and (x_j, y_j) lies perpendicular to the location described by (ρ', θ') or $(-\rho', \theta' + 180°)$ in the spatial domain. Notably, only one intersection is required to describe the line's location in the spatial domain. The range of θ can be reduced using the Glide Reflection, as described in Section 3.2.1. Reducing the range of θ has the effect of halving the memory required to store the $\rho\theta$-plane.

3.2.1 The Glide Reflection

A glide reflection (also known as a transflection) is the reflection and translation of a two-dimensional object. The object is reflected across a line, and a translation is applied parallel to the line of reflection. Upon inspecting the $\rho\theta$-plane in Figure 3.3, it can be seen that each sinusoid is periodic across intervals of 360°. A $\rho\theta$-plane contains redundancy if any point can be mapped to another using the glide reflection relation,

$$\rho(\theta) = -\rho(\theta + 180^\circ). \tag{3.3}$$

The range of θ can be reduced to $[0, 180^\circ)$ as shown in Figure 3.4. The work in this thesis exploits the glide reflection to minimise the resource consumption required for storing the $\rho\theta$-plane and increase computational performance. Further information about redundancy in the $\rho\theta$-plane is given in [79].

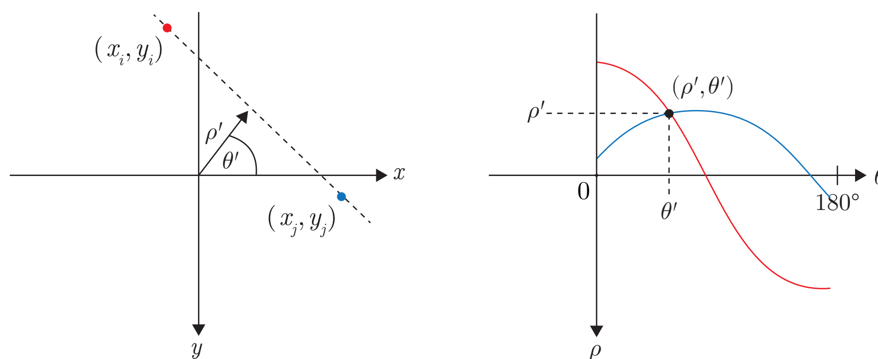


Figure 3.4: Points (x_i, y_i) and (x_j, y_j) in the spatial domain (left) are mapped to the $\rho\theta$-plane (right) using (3.2). The size of the $\rho\theta$-plane is halved using the glide reflection given in (3.3).

For the remainder of this thesis, the term HPS will refer exclusively to the $\rho\theta$-plane and is denoted mathematically as $A(\rho, \theta)$. Additionally, the LHT described by Duda & Hart [3] will be referred to as the standard LHT.

3.2.2 Memory Requirements

In this section, an expression is derived that calculates the HPS memory requirements for a given image resolution. Memory is measured using bits, representing a technology-independent unit for comparing memory consumption between embedded devices.

As previously mentioned in Section 3.1, the HPS is normally divided into cells (also known as bins) that accumulate votes. The HPS stores votes in discrete locations quantised over N_ρ and N_θ levels, where N_ρ is the number of levels along the ρ-axis and N_θ is the number of levels along the θ-axis. The ρ and θ axes each have their own regular discrete step values, δ_ρ and δ_θ, respectively. As previously defined by the glide reflection (Section 3.2.1), the operational range of θ is between $[0°, 180°)$. The range of the ρ-axis is determined by the maximum displacement of a line. To find the range of the ρ-axis, the image origin must first be decided. Consider an $M \times N$ image, where the length of the image diagonal D is given by the expression

$$D = \sqrt{M^2 + N^2}. \tag{3.4}$$

The location of the image origin impacts the total memory required by the HPS, as it affects the total number of quantisation levels across the ρ-axis. Figure 3.5 presents an example of the LHT when the image origin is at the top left corner of the image.

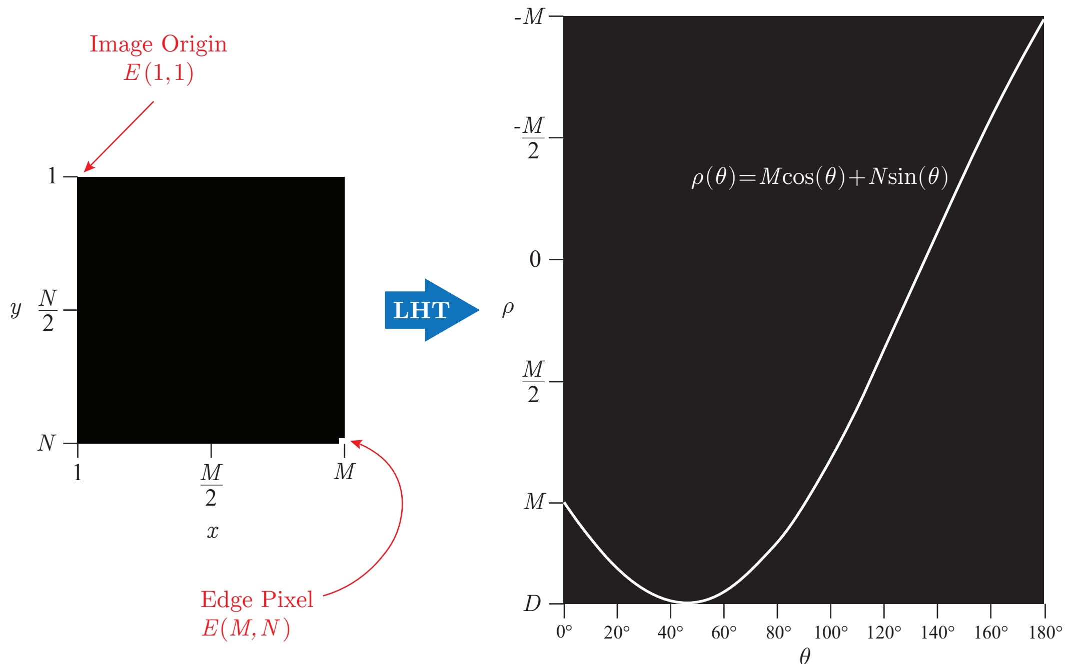


Figure 3.5: The spatial image domain of $M \times N$ pixels, containing one edge pixel at the bottom right corner, which has been enlarged for visualisation purposes (left). The corresponding HPS containing votes that have been mapped using (3.2), presented in top-down view (right). The image origin is located at the top left corner of the image.

The example from Figure 3.5 applies the LHT to the edge image over a discrete set of θ values in the range $[0°, 180°)$. The resulting HPS exhibits an operational range across the ρ-axis of $[-M, D]$. This range is computed by plotting the Hough parameters of the edge pixel furthest away from the image origin. In comparison, when the image origin is placed in the centre of the image, the range of the ρ-axis is $[-D/2, D/2]$, which is considerably smaller. Therefore, memory consumption of the HPS can be optimised by positioning the image origin in its centre. A full example demonstrating this configuration is given in Section 3.2.3, where the ρ-axis of the HPS operates across regular discrete intervals, δ_ρ, in the range $[-D/2, D/2]$.

The third dimension of the HPS, 'votes', is limited by the number of pixels that represent the longest possible line in the candidate image. Therefore, votes are in the range $[0, D]$. The total memory bits required by the HPS for an $M \times N$ image can now be derived. Each cell in a HPS of size $N_\rho \times N_\theta$ must be represented by $\lceil \log_2(D) \rceil$ bits. Therefore, the total number of memory bits required to store the HPS is given as

$$b = N_\rho N_\theta \lceil \log_2(D) \rceil. \tag{3.5}$$

Chapters 4, 5, and 6 use (3.5) to compute memory consumption and enable comparisons with LHT architecture designs.

3.2.3 Vote Accumulation

Figure 3.6 presents an edge image of 100×100 pixels and its associated HPS for accumulating votes. As described in Section 3.2.2, the image origin is placed centrally to optimise the memory required to store the HPS. The HPS is divided into 144×180 accumulator cells, using $\delta_\rho = 1$ and $\delta_\theta = 1°$. As shown, the candidate image consists of five edge pixels labelled 1 to 5. There is an edge pixel at each corner of the image, and one approximately in the centre of the image. The five edge pixels have been processed using (3.2) over a discrete set of θ values. The corresponding parameters (ρ, θ) for each edge pixel have accumulated in the HPS to produce four sinusoids, and a unique case where a horizontal line has formed due to the centre edge pixel.

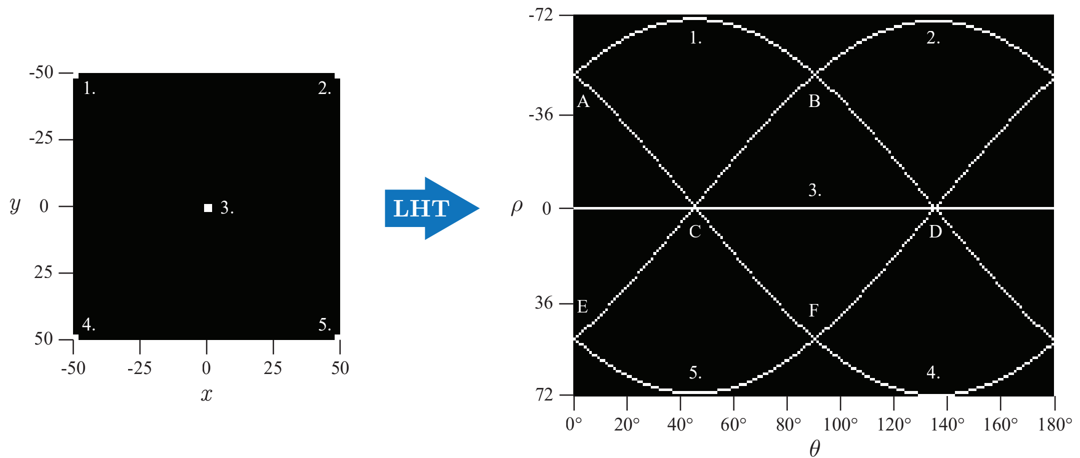


Figure 3.6: The spatial image domain of 100×100 pixels, containing five edge pixels, which have been enlarged for visualisation purposes (left). The corresponding HPS containing votes that have been mapped using (3.2), presented in top-down view (right).

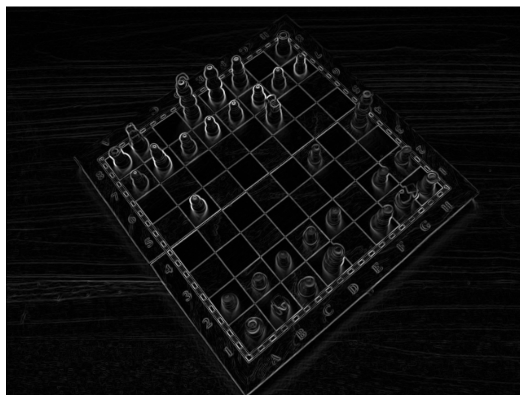
As shown in Figure 3.6, each edge pixel has been labelled 1 to 5 so that the corresponding sinusoid in the HPS can be identified easily. Each corner edge pixel has been mapped to a sinusoid, while the centre pixel only produces a horizontal line. The centre pixel coordinates are $x = 0$, and $y = 0$. Therefore, when substituting these values into (3.2) across all values of θ, the corresponding magnitude of displacement for each orientation will be zero, resulting in a horizontal line.

The HPS shown to the right of Figure 3.6 also contains labels A to F, which are not to be confused with mathematical symbols and variables in this thesis that use an italic font. These labels each indicate points of collinearity between the edge pixels in the spatial domain. Notice that labels C and D indicate the points of greatest collinearity. At point C, edge pixels 2, 3, and 4 are collinear. Point D shows the collinearity of edge pixels 1, 3, and 5.

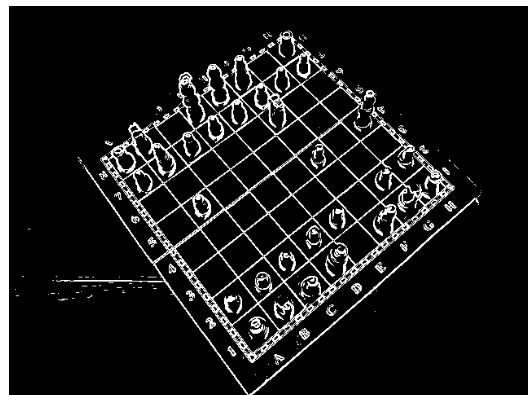
Edge images generally contain hundreds and thousands of edge pixels that all need to be mapped to the HPS. Consider the 640×480 pixel image of a chessboard shown in Figure 3.7a. Initially, the colour image is converted to greyscale as shown in Figure 3.7b. Edge detection is then performed using Sobel operators (Figure 3.7c) and thresholding to produce the edge image seen in Figure 3.7d.

(a) Colour image of a chessboard.

(b) Greyscale image of a chessboard.



(c) Gradient magnitude image of a chessboard using Sobel operators.

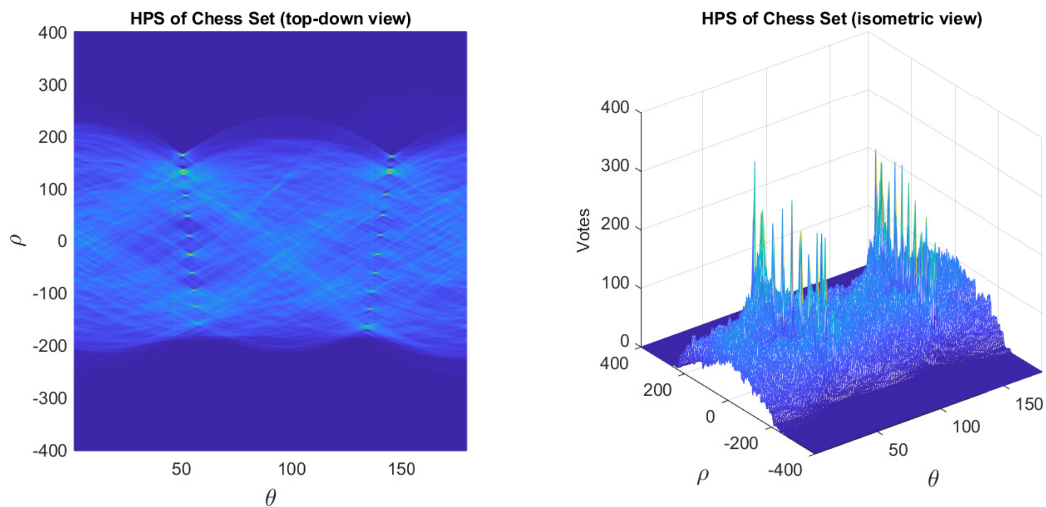


(d) Edge image of a chessboard using Sobel operators and thresholding ($T = 100$).

Figure 3.7: A colour image of a chess board (a) that has been converted to a greyscale image (b). Sobel operators are applied to the greyscale image to produce the gradient magnitude image (c) and thresholding is applied to produce an edge image (d).

The edge image shown in Figure 3.7d contains 24,944 edges. When there are 180 orientations of θ, each edge will vote in the HPS 180 times, resulting in a total of 4,489,920 votes. The corresponding HPS for the chessboard edge image can be seen in Figure 3.8. It was generated by operating θ over the range $[0°, 179°]$, and setting $\delta_\theta = 1°$ and $\delta_\rho = 1$. The HPS is presented using two different views; top-down and isometric.

The top-down view of the HPS, presented in Figure 3.8a, can be used to determine areas of high collinearity. Collinear parameters can be identified from ribbon bow shapes that are formed by accumulating sinusoids of different phases and amplitudes. At the centre of each ribbon bow are a significant number of votes that accumulate to form a local peak, or a global maximum, in the HPS.



(a) Top-down view of the chessboard HPS.

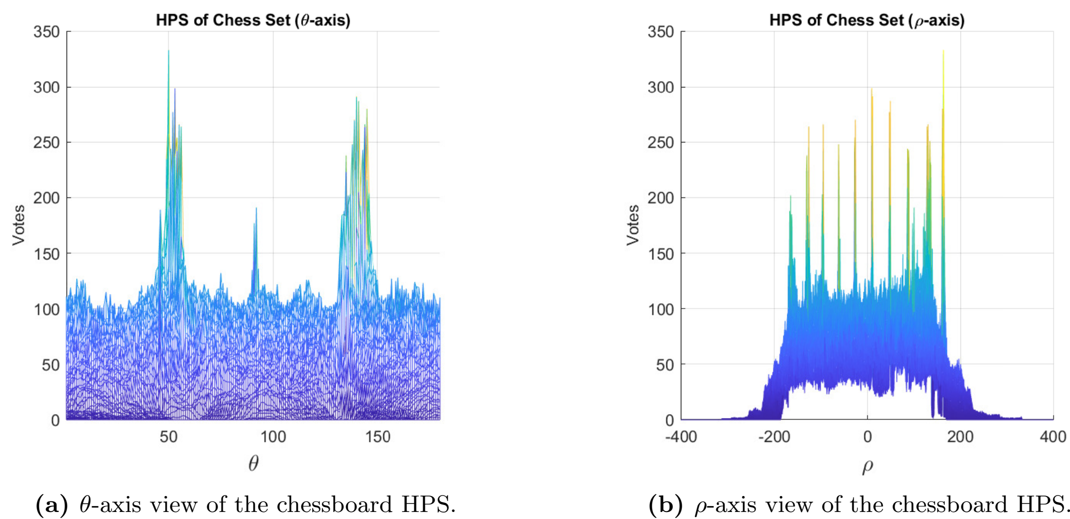
(b) Isometric view of the chessboard HPS.

Figure 3.8: The HPS for the chessboard edge image in Figure 3.7d. The HPS is shown from the top-down (a) and isometric view (b).

Figure 3.8b presents an isometric view of the HPS, which is useful for inspecting the distribution of peaks across the θ and ρ axes. For example, the number of votes accumulated by a single peak can be easily determined. Two alternative ways of inspecting the HPS are given in Figure 3.9.

The view of the θ-axis, shown in Figure 3.9a, helps establish the most common orientations of lines in the candidate image. This view is useful if attempting to reduce the HPS memory consumption or total algorithm computation. It can also help extract the orientation of an entire object. For example, the θ-axis view of the HPS can be used to determine the rotation of the chessboard from the horizontal midpoint of the candidate edge image.

The plot in Figure 3.9b presents the ρ-axis of the HPS, which helps analyse the distribution of edge pixels and collinear features in the candidate image. Notice that most votes are in the centre of the plot. This distribution corresponds to a high density of edge pixels residing around the origin of the candidate image. Processing speed and memory allocation could be improved by cropping the input image before applying the LHT. However, depending on application requirements, image cropping may impact the accuracy of line detection.



(a) θ-axis view of the chessboard HPS.

(b) ρ-axis view of the chessboard HPS.

Figure 3.9: The HPS for the chessboard edge image in Figure 3.7d. The HPS is shown from the θ-axis (a), and ρ-axis (b).

This section has presented vote accumulation in the HPS for the chessboard image. Peak searching and line reconstruction of the chessboard image is demonstrated in the next section.

3.2.4 Line Reconstruction

When voting is complete, the HPS is searched for parameters that have accumulated peaks. If a peak has accumulated enough votes to be larger than a predefined threshold, then the corresponding parameters are selected for reconstruction. These parameters are passed into an inverted LHT kernel as given in (3.6) and (3.7), where $\theta \in [0°, 180°)$.

$$x = \frac{\rho - y\sin(\theta)}{\cos(\theta)}, \quad \theta \neq 90° \tag{3.6}$$

$$y = \frac{\rho - x\cos(\theta)}{\sin(\theta)}, \quad \theta \neq 0° \tag{3.7}$$

The inverse LHT kernel is applied by selecting (3.6) or (3.7) so that θ does not produce an undefined result. For instance, if $\theta = 90°$, then (3.7) would be selected as (3.6) produces an undefined result. For an $M \times N$ image, equation (3.6) is applied over a discrete range of y values in the range $[-N/2, N/2)$. This operation produces corresponding x values that can be used to determine the locations of linear elements

in the original image. Similarly, (3.7) is applied over a discrete range of x values in the range $[-M/2, M/2)$ to produce corresponding y values. In this thesis, line reconstruction will only be used to validate the location of lines in the original image.

The HPS of the chessboard image in Figure 3.8 will now undergo thresholding to demonstrate line reconstruction. Note that the threshold value does not impact the memory consumption of the HPS. The largest peak in the HPS has accumulated 333 votes. Therefore, an arbitrary threshold of 200 votes is selected to suppress unwanted parameters, which is approximately 60% of the maximum peak. After thresholding, the Hough parameters corresponding to the remaining peaks are extracted and used in (3.6) and (3.7) to reconstruct lines. The reconstructed image is given in Figure 3.10a.

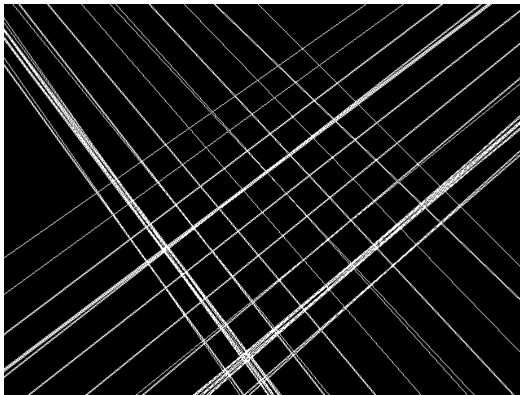
Notice that the reconstructed image in Figure 3.10a contains infinite lines, i.e. lines that end at the borders of the image. To correct infinite lines, the reconstructed image is combined with the original edge image using a logical AND operation, as in Figure 3.10b. This image contains collinear features of the chessboard without infinite lines and chess pieces obfuscating the chessboard.

3.3 Literature Review

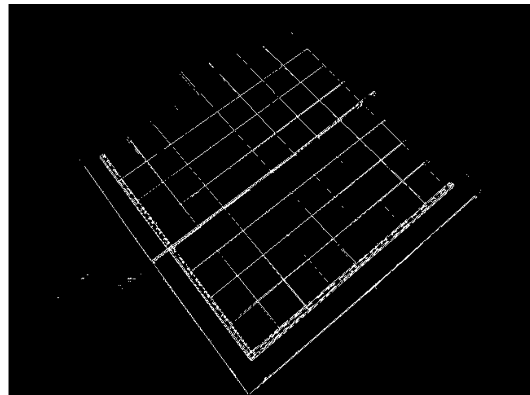
There are many optimisations described in the literature that improve the processing time, line extraction accuracy, and resource allocation of the LHT. This section presents a literature review of the LHT that is separated into three subsections: software designs and optimisations, multiplierless architectures, and resource-efficient architectures. Previously published literature that is relevant to the work described in this thesis is highlighted.

3.3.1 Software Implementations

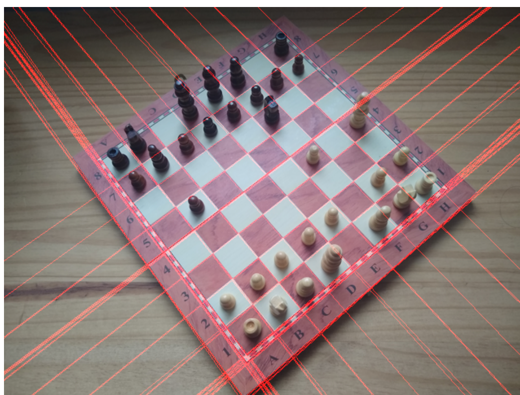
This section reviews software implementations of the LHT, which include the Gradient LHT, the Adaptive Hough Transform, the Fast Incremental Hough Transform 2, the memory-compressed Hough Transform, random sampling and probabilistic Hough Transforms, and the Kernel Hough Transform.



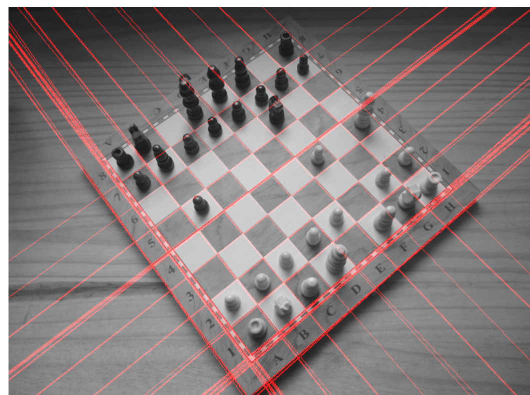
(a) Reconstructed line image of the chessboard created using (3.6) and (3.7).



(b) Reconstructed image combined with the original edge image using a logical AND operation.



(c) Overlay of the reconstructed image and the original colour image.



(d) Overlay of the reconstructed image and the original greyscale image.

Figure 3.10: Infinite line reconstruction in (a), finite line reconstruction in (b), overlay of the reconstructed image and colour image in (c), and overlay of the reconstructed image and greyscale image in (d).

The Gradient LHT

The computation of the LHT was significantly reduced by O'Gorman & Clowes [10]. The authors approximated the gradient orientation of edge pixels to reduce the number of votes applied to the HPS. This technique is named the Gradient LHT and can reduce the number of arithmetic operations. The gradient orientation, α, is calculated using (2.11) for each edge pixel. The gradient orientation image of the chessboard is shown on the left of Figure 3.11, and a magnified region can be seen on the right. Notice that collinear features are constructed of pixels that share the same gradient orientation.

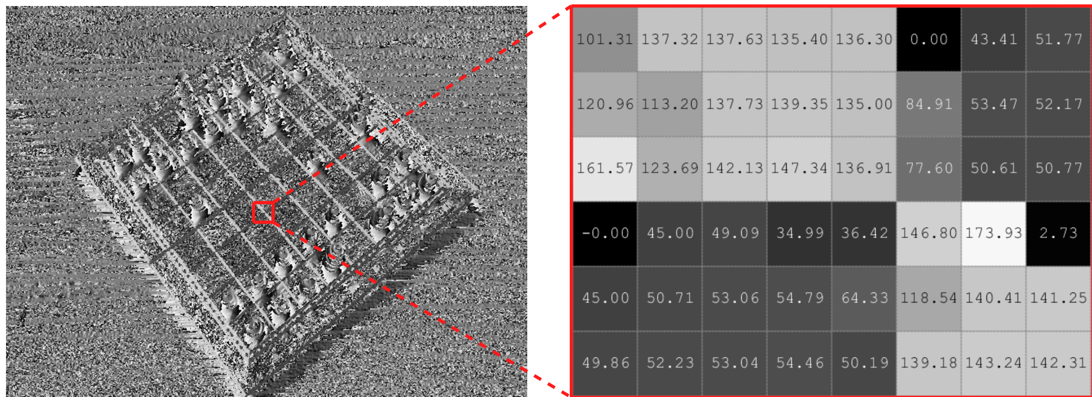
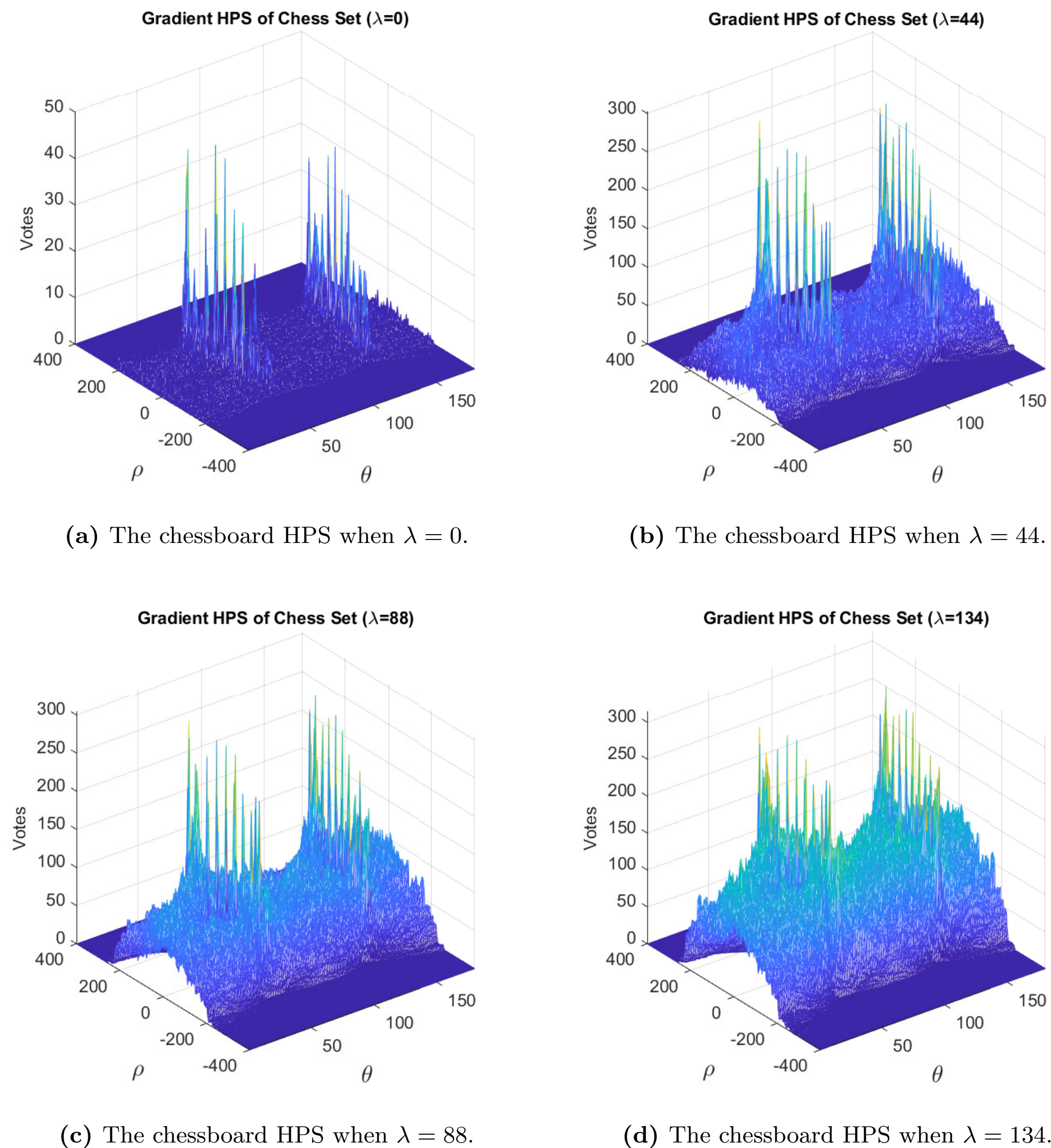


Figure 3.11: The gradient orientation image of the chessboard (left), and a magnified region (right). The intensity of each pixel corresponds to its gradient orientation value.

The parameters (ρ, α) for each edge pixel are used to apply a single vote in the HPS. To improve the sensitivity of line detection, the number of votes can be increased by distributing votes evenly around α such that $[\lambda/2-\alpha : \alpha : \lambda/2+\alpha]$, where λ determines the number of additional votes applied to the HPS per edge pixel. The effects of varying λ can be seen in the HPS produced using the Gradient LHT. Figure 3.12 presents the HPS of the chessboard when $\lambda = [0, 44, 88, 134]$, $\delta_\theta = 1°$ and $\delta_\rho = 1$.

Figure 3.12a presents the HPS when there is one vote per edge pixel. This voting scheme reduces the sensitivity of line detection as compared to the standard LHT, due to edge pixels that are not entirely collinear. These edge pixels cause peaks to form in neighbouring bins within the HPS rather than the primary bin of the associated line. To improve the sensitivity of line detection, the number of votes applied to the HPS per edge pixel can be increased as in (b), (c), and (d) in Figure 3.12.

Setting $\lambda = 0$ will not cause significant accuracy issues when extracting peaks in the HPS. However, there may be a substantial drop in line detection compared to using a larger value of λ. One vote per edge pixel may be a suitable voting scheme for applications that do not require significant line detection accuracy. However, if line detection accuracy is important for an application, it can be traded off with the computational requirements of the Gradient LHT by increasing λ. O'Gorman & Clowes [10] mention that setting $\lambda = 20$ is suitable for most applications. The Gradient LHT is of particular importance to the work presented in Chapter 6 of this thesis.



(a) The chessboard HPS when $\lambda = 0$.

(b) The chessboard HPS when $\lambda = 44$.

(c) The chessboard HPS when $\lambda = 88$.

(d) The chessboard HPS when $\lambda = 134$.

Figure 3.12: Four HPS results for the chessboard edge image in Figure 3.7d, using the Gradient LHT. The HPS when $\lambda = 0$ (a), $\lambda = 44$ (b), $\lambda = 88$ (c), and $\lambda = 134$ (d).

The Adaptive Hough Transform

Illingworth & Kittler introduce the Adaptive Hough Transform (AHT) in [12]. The AHT uses a coarse-to-fine approach of extracting lines from an image. The HPS is initially coarsely quantised with large discretisation steps. The candidate image is then processed by (3.2) and votes are accumulated in the HPS. Peaks that have accumulated a significant number of votes are considered to be regions of high collinearity. The

peaks then undergo a second round of processing using a finely quantised HPS. The process is repeated until a predefined target resolution is achieved, which is sufficient to accurately extract linear features. The AHT algorithm can be used to reduce the storage requirements of the HPS, as the same memory is used to implement the coarse and fine accumulators. Furthermore, computation may be reduced in comparison to the original LHT, depending on the number of iterations performed.

The AHT is a multi-pass algorithm that scans the input image several times before producing an output result. These type of algorithms are highly suited to software implementations, as general purpose processors can easily buffer arrays of memory. FPGA architectures can also buffer images in external memory or on-chip memory. However, these architectures require sophisticated memory control. A significant disadvantage of using an FPGA architecture of the AHT, in comparison to the LHT, is the amount of time required to scan the input image multiple times. This process causes significant delay when computing the output HPS.

The Fast Incremental Hough Transform 2

The Fast Incremental Hough Transform 2 (FIHT2) algorithm proposed by Koshimizu *et al.* [80], is able to detect lines in digital images without using trigonometry or performing multiplications. FIHT2 employs incremental equations that use the value of $\rho(\theta_n)$ from a previous calculation to generate the value of $\rho(\theta_{n+1})$. The starting point for deriving the FIHT2 algorithm is

$$\rho(\theta_n) = \begin{cases} x\cos(\theta_n) + y\sin(\theta_n) + \dfrac{\varepsilon x \sin(\theta_n)}{2}, & \text{if} \quad (0 \leq n < N_\theta/2), \\ x\cos(\theta_n) + y\sin(\theta_n) + \dfrac{\varepsilon y \cos(\theta_n)}{2}, & \text{if} \quad (N_\theta/2 \leq n < N_\theta), \end{cases} \tag{3.8}$$

where $\varepsilon = \pi/N_\theta$ and $n = 1, 2, \ldots, N_\theta - 1$. The authors use the first derivative of (3.8) with respect to the variable θ_n to produce $\rho'(\theta_n)$. The FIHT2 incremental equations for a pixel defined by coordinates (x, y) are given as

$$\begin{aligned} \rho(\theta_{n+1}) &= \rho(\theta_n) + \varepsilon\rho'(\theta_n), \quad \text{for} \quad (0 \leq n < N_\theta/2), \\ \rho'(\theta_{n+1}) &= \rho'(\theta_n) - \varepsilon\rho(\theta_{n+1}), \quad \text{for} \quad (N_\theta/2 \leq n < N_\theta). \end{aligned} \tag{3.9}$$

Each incremental equation can be conveniently initialised using $\theta_0 = 0°$, which corresponds to $\rho(0°) = x$ and $\rho'(0°) = y$. The authors also demonstrate that careful selection of ε can greatly reduce computation. If ε is set to 2^{-m}, where m is a natural number, the multiplication operations in (3.9) can be replaced by simple shift operations. The new FIHT2 incremental equations become,

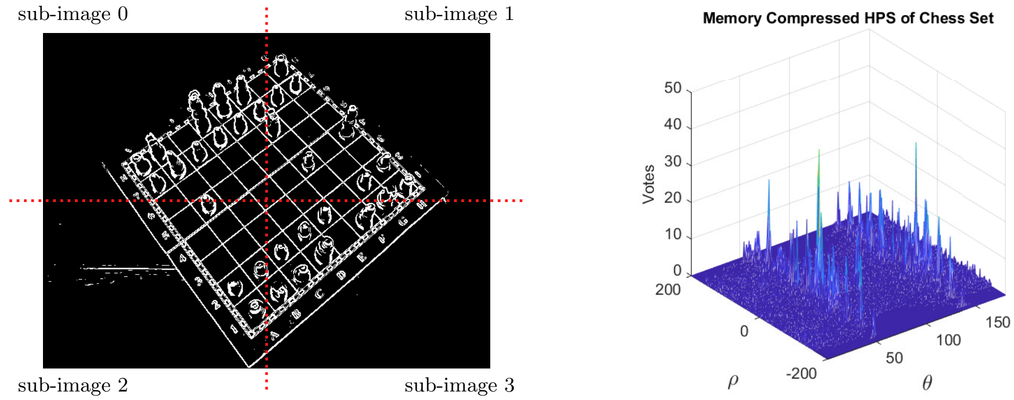
$$\begin{aligned} \rho(\theta_{n+1}) &= \rho(\theta_n) + 2^{-m}\rho'(\theta_n), \quad \text{for} \quad (0 \le n < N_\theta/2), \\ \rho'(\theta_{n+1}) &= \rho'(\theta_n) - 2^{-m}\rho(\theta_{n+1}), \quad \text{for} \quad (N_\theta/2 \le n < N_\theta). \end{aligned} \tag{3.10}$$

For example, if $\varepsilon = 1/128$, then the number of discretisation levels across the θ-axis becomes $N_\theta = 100$. The incremental equations in (3.10) demonstrate that only shift and add operations are required to compute the Hough parameters. Although this technique removes multiplication operations and trigonometry, the design is restricted to particular values of both N_θ and discrete step δ_θ.

Memory-Compressed LHT

Ser & Siu [23] describe a technique that uses lossy compression of the HPS to reduce memory requirements. Initially, the candidate edge image is partitioned into many equal-sized sub-images. This action has the effect of reducing the size of the HPS along the ρ-axis. The gradient orientation is used to reduce the number of votes accumulated in the HPS per edge pixel, where $\lambda = 0$. Each sub-image accumulates votes in the same HPS, which may cause peaks to merge together. However, the HPS is now reduced in size by a factor of $\sqrt{K_\rho}$, where K_ρ is the total number of equal-sized sub-images that must be a power of 4. Figure 3.13 presents the memory-compressed HPS and the chessboard edge image, which has been partitioned into four equal-sized sub-images.

Another memory array, known as a Region Bitmap (RBM), is introduced to maintain a record of the Hough parameters (ρ, α) for all edge pixels in the image. There is one RBM for each sub-image of the original edge image. The RBM can be used to identify peaks in the HPS that correspond to each sub-image and attempt to separate these peaks for line reconstruction. The RBM for each sub-image is an array of size $N_\theta \times N_\rho/\sqrt{K_\rho}$, where each location in the array uses 1 bit to record (ρ, α).



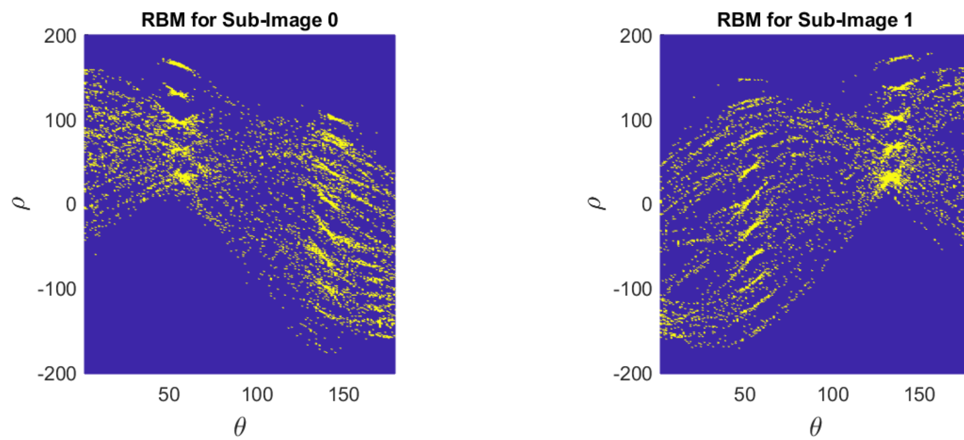
(a) Edge image of the chessboard using Sobel operators and thresholding ($T = 100$). The number of sub-images is $K_\rho = 4$.

(b) Memory-compressed HPS of the chessboard edge image, which is halved in size across the ρ-axis in comparison to Figure 3.8a.

Figure 3.13: The sub-images of the chessboard edge image in (a), are used to vote in the memory-compressed HPS in (b).

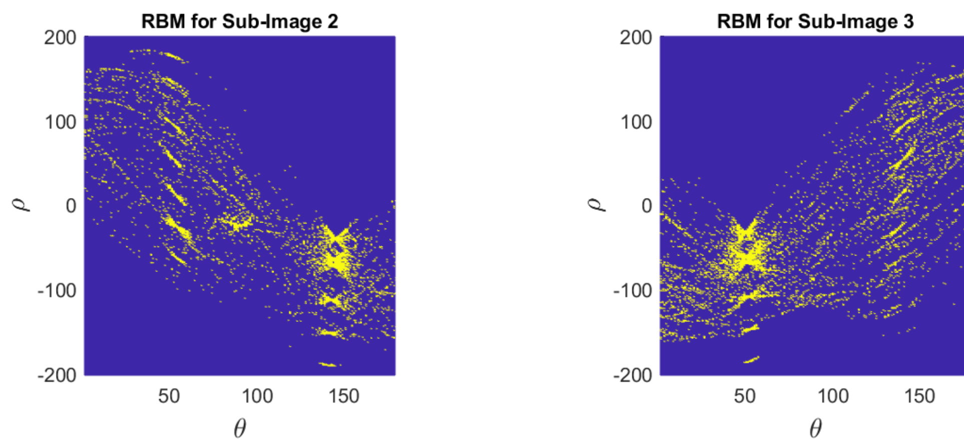
Figure 3.14 illustrates four RBM memory arrays for each chessboard sub-image. After voting is complete, the peaks in the HPS are extracted and the RBMs for each sub-image are analysed. Peaks that have not merged in the HPS can be easily identified as they have only been recorded by one RBM. Merged peaks are more complicated and require a dedicated peak separation algorithm. The authors of [23] describe a technique to separate merged peaks in the HPS. A local filter with a 5×5 window is used to identify clusters of bits in each RBM around the identified peak. The RBM that contains the highest cluster of bits is awarded the votes in the HPS for the corresponding peak.

A significant problem with the memory-compressed HT approach is the detection of spurious lines. Errors occur when the candidate edge image contains high levels of noise. The noise is recorded on the RBM, which can prevent the peak separation algorithm from operating correctly. However, the algorithm is very successful when the candidate image contains low levels of noise and lines. This thesis presents an original contribution to knowledge, which is a modified version of the memory-compressed LHT described above. The new memory compression technique is modified for FPGA implementation and named the Angular Regions LHT (ARLHT). The ARLHT algorithm and FPGA architecture is described in Chapter 6 of this thesis. The performance of the ARLHT and its limitations are also discussed.



(a) The RBM for sub-image 0 of the chessboard.

(b) The RBM for sub-image 1 of the chessboard.



(c) The RBM for sub-image 2 of the chessboard.

(d) The RBM for sub-image 3 of the chessboard.

Figure 3.14: Four RBM memory arrays for each sub-image of the chessboard edge image.

Random Sampling and Probabilistic Methods

There are techniques that use random sampling to improve the performance of the LHT. Kiryati *et al.* [81] describe a Probabilistic Hough Transform (PHT) for the extraction of lines, which uses randomly selected edges to reduce processing time. The PHT operates on a limited number of randomly sampled edge pixels from the input image. The number of edge pixels processed by the PHT is known as the poll size. The authors provide experimental data where several candidate edge images are processed using the PHT. Results demonstrate that random sampling has the effect of reducing computation with few false detections, provided that the poll size is large enough.

The PHT was extended by Matas *et al.* in [82] where a Progressive Probabilistic Hough Transform (PPHT) is described. The authors explain that the PHT is only effective if the length of the line is already established, which in turn determines the necessary poll size. This information is not typically known in many line detection applications. Therefore, the authors describe a different mechanism to determine the effective poll size during algorithm operation. The PPHT algorithm dynamically controls the detection of a line by considering the total number of votes cast during operation and the highest peaks in the HPS. This technique is advantageous as the algorithm can be interrupted during operation and still provide useful results. Furthermore, the algorithm can be constrained to detect a specific length of line before halting execution. At the time of writing, the PPHT is a supported algorithm in the Open Computer Vision (OpenCV) library [83], demonstrating its overall acceptance by the image processing community.

The Kernel Hough Transform

Fernandes & Oliveira [11] present the Kernel Hough Transform (KHT), which addresses the computational complexity of the LHT by carefully selecting clusters of collinear pixels for voting. Each cluster is convolved with an elliptical-Gaussian kernel and the result is used to apply votes in the HPS using a custom voting procedure. The authors performed several experiments where they applied the KHT to a set of candidate test images using an AMD Athlon 64 3700+ (2.21 GHz) computer with 2 GB of RAM. The proposed algorithm was reported to achieve up to 200 frames per second (fps) for images containing 512×512 pixels.

3.3.2 FPGA Architectures and Development Tools

FPGAs are often selected to accelerate the LHT, as they offer high computational performance and low latency processing by exploiting their parallel processing capabilities. This section reviews several FPGA architectures of the LHT from previously published works. These architectures include multiplierless and resource-efficient designs, which are of significant relevance to the novel architectures presented in Chapters 5 and 6.

Multiplierless Architectures

Tagzout *et al.* [84] implemented the FIHT2 algorithm on a XC4008EPC84 FPGA. The FIHT2 algorithm is particularly suited for implementation on this FPGA as it does not require any specialised DSP resources. The authors found that a quantisation error caused by approximations of $\sin(\theta)$ and $\cos(\theta)$ leads to erroneous results after several iterations. The authors describe how to continuously correct the error by tolerating a limited amount of error in the incremental equations. When the error grows larger than the tolerance, the error is removed by recalculating (3.2).

CORDIC is an efficient technique for implementing trigonometric functions and performing simple math operations in an FPGA. Karabernou & Terranti [85] combine the Gradient LHT and CORDIC to minimise FPGA resource consumption on a XC4010EPC84 device. A functional block diagram of this system is presented in Figure 3.15. The authors use Circular CORDIC configured in vectoring mode to compute the gradient orientation, and Circular CORDIC configured in rotation mode to compute (3.2). Linear CORDIC is also used during the line reconstruction process to compute the inverted LHT equations in (3.6) and (3.7).

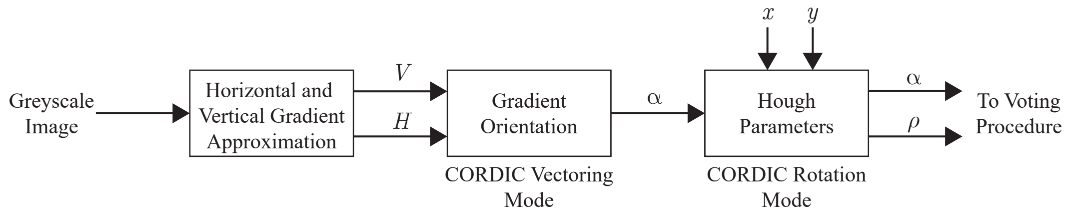


Figure 3.15: Overview of an FPGA architecture of the LHT that primarily uses CORDIC [85].

Lee & Evagelos [86] describe the Hybrid-Log Hough Transform, an innovative approach to implementing the LHT on a Virtex-4 device without using multipliers. Their architecture performs linear multiplications in the log domain as simple addition operations. An example of the hybrid-log Hough kernel that computes (3.2) can be seen in Figure 3.16. The architecture boasts excellent resource consumption and timing performance. However, the authors allude to accuracy issues when using this technique, and these can be seen in their experimental results when compared to the HPS produced using (3.2).

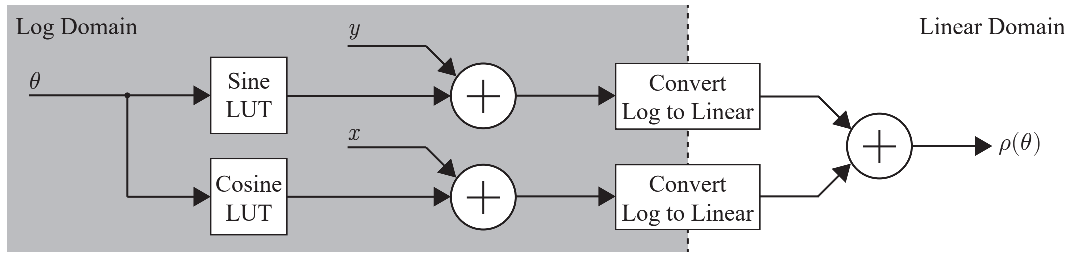


Figure 3.16: An illustration of the Hybrid-Log Hough kernel for computing the Hough parameters.

Lu *et al.* [13] designed an unrolled LHT architecture that uses shift and add operations to compute the Hough parameters. The advantage of this architecture is that it does not use any multiplication resources. However, its design has significant limitations, such as its high latency and fixed value of δ_θ, which is 0.8952°. Although not mentioned by the authors, their LHT algorithm is very similar to the FIHT2 algorithm. An illustration of the unrolled LHT architecture is presented in Figure 3.17. Notice that it contains iterative cells to compute the Hough parameters, which is similar to the CORDIC algorithm given in Appendix A.

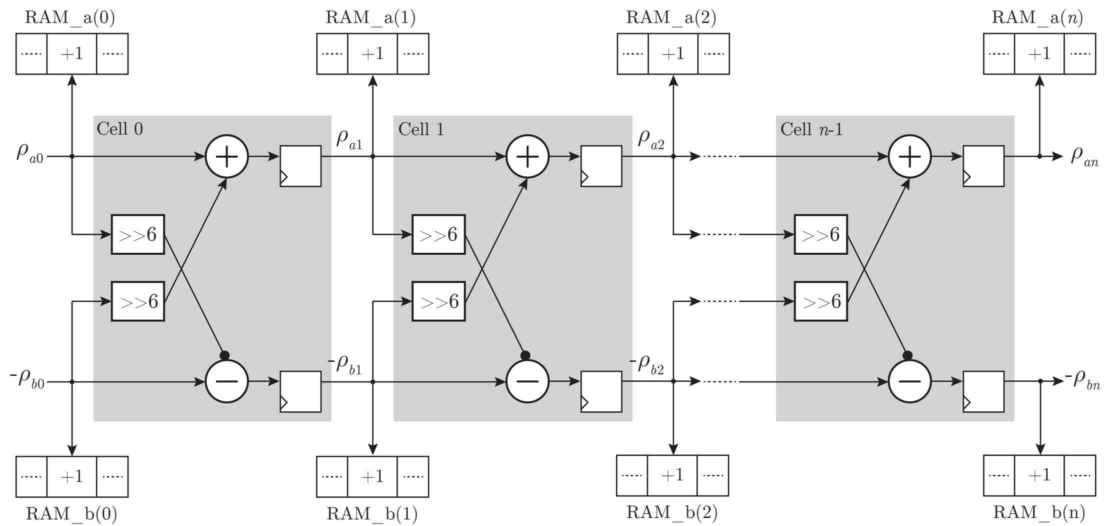


Figure 3.17: An unrolled LHT architecture that uses shift and add operations to compute the Hough parameters. The variables ρ_a and ρ_b are used to denote signal paths in the architecture design. See the corresponding publication in [13] for more information.

Resource-Efficient Architectures

The LHT equation in (3.2) requires two multipliers and one adder to compute ρ for one angle in θ. If computing the LHT across 180 values of θ in the range $[0°, 180°)$, then 360 multiplications and 180 additions would be required. Zhou *et al.* [14] developed an efficient parallel LHT architecture that requires two multiplications and two additions to compute two angles in θ. This reduction in complexity is possible by partitioning θ into two ranges $[0, 90)$ and $[90, 180)$. The LHT for each range of θ can be computed at the same time by recognising that $x\cos(\theta) = -x\cos(180° - \theta)$ and $y\sin(\theta) = y\sin(180° - \theta)$. An FPGA architecture of the Hough kernel that uses this technique to simultaneously compute $\rho(\theta)$ and $\rho(180° - \theta)$ is illustrated in Figure 3.18.

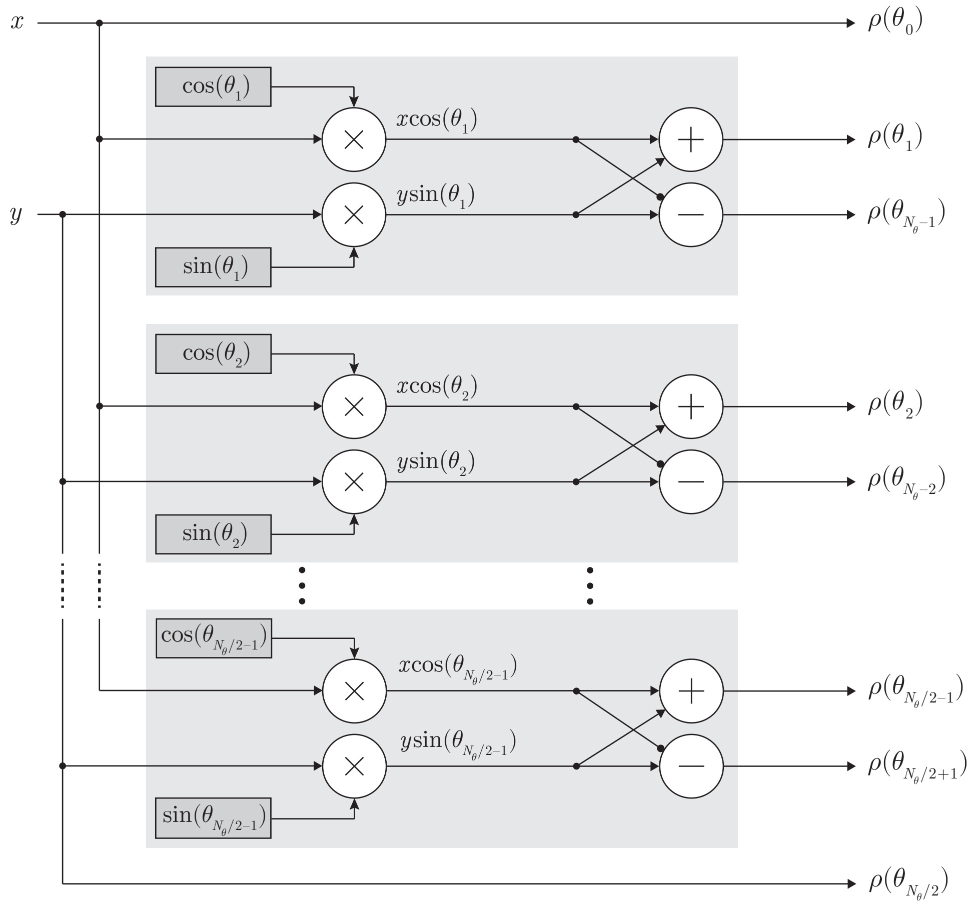


Figure 3.18: A Hough kernel that can compute $\rho(\theta)$ and $\rho(180° - \theta)$ using two multipliers and two adders. This architecture assumes $\theta_0 = 0°$, $\theta_{N_\theta/2} = 90°$, and N_θ is an even number.

Additionally, the authors recognised that $x\cos(0°) = x$, $x\cos(90°) = 0$, $y\sin(0°) = 0$, and $y\sin(90°) = y$, which further reduces multiplication requirements. Overall, their resource-efficient LHT architecture is able to process a 512×512 pixel image using 178 DSP48E1 slices and 90 BRAMs in an AMD Virtex-6 device.

In similar work, Zhou *et al.* [20] described a Gradient LHT architecture that consumes 13 DSP48E1 slices in a Virtex-7 device. The voting range, λ, is set to 8, significantly reducing the number of multiplications required by the architecture. The authors also exploit the raster scan streaming of image data to pre-compute $y\sin(\theta)$ for the next image row and store the values in registers. This technique significantly reduces resource allocation, allowing $y\sin(\theta)$ to be computed using a single DSP48E1 slice. The authors do not assign a name to their architecture. Therefore, the architecture will be referred to as the Look-Ahead Kernel for the remainder of this thesis. An illustration of the Look-Ahead Kernel is shown in Figure 3.19.

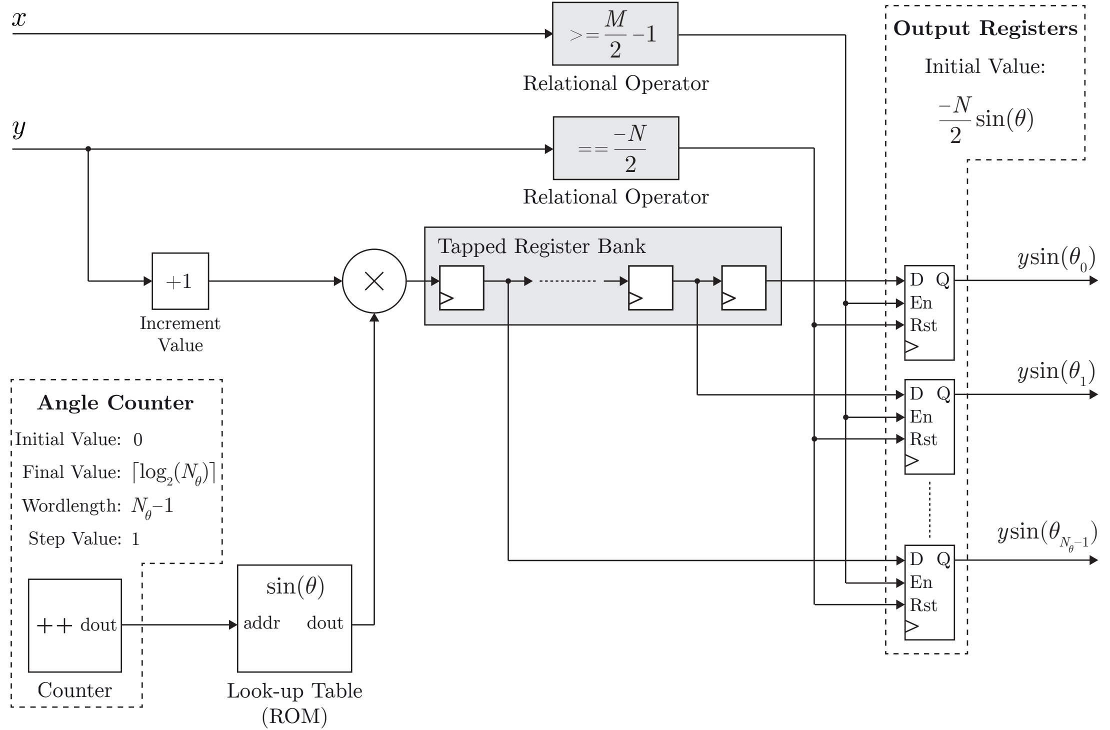


Figure 3.19: An FPGA architecture of the Look-Ahead Kernel, which calculates $y\sin(\theta)$ after each image row and stores the results in N_θ registers. Control logic has been removed from this design to simplify the illustration.

Lastly, Zhou *et al.* [87] combine the architectures presented in [14] and [20] to create an efficient parallel LHT design that consumes 90 DSP48E1 slices in a Virtex-6 device. A significant contribution of this thesis, presented in Chapter 5, improves upon this work by reducing DSP slice requirements by 50% and optimising the memory consumption of the HPS.

Elhossini & Moussa [88] present two FPGA architecture designs that implement the LHT and the Circle Hough Transform (CHT), which detects circles in a digital image. The authors claim that their architectures are memory efficient and do not use external memory. The memory efficient design is achieved by simply increasing the discretisation step of θ, such that $\delta_\theta = 11.25°$. The authors do not describe any other algorithmic or architectural techniques to reduce memory consumption.

Chen *et al.* [89] describe a novel technique of implementing the HPS in external memory, which reduces on-chip memory requirements. The input edge image is first divided into several sub-images, and run-length encoding is performed to reduce processing requirements. Each encoded sub-image is then read into the LHT architecture, which uses the FIHT2 algorithm to compute the Hough parameters. The parameters are then used to vote in a local accumulator. Once voting is complete, the local accumulator is transferred to external memory so that votes are accumulated into the HPS. This architecture is an excellent example of using external memory to store the HPS. However, there are several disadvantages to deploying this architecture. The speed and bandwidth of the external memory limit the execution time of the architecture. The architecture is non-deterministic in terms of processing time, causing video frames to be processed on a best-effort basis. If the architecture is modified for higher resolutions, the memory bandwidth will also need to increase, which may not be possible due to technical limitations.

Bailey [21] describes a novel technique to reconstruct lines from an HPS stored in on-chip FPGA memory. After voting, the accumulator memory is scanned for peaks and reset during the vertical blanking period of a video stream. If a peak is found, an additional memory bit in the accumulator is used as a ‘peak flag’ to indicate the presence of a peak. As the next video frame is streamed into the LHT design, the

pixel coordinates are converted to Hough parameters that index the accumulator, subsequently reading the peak flag from memory. If the peak flag is set, then the current pixel of the output frame contains a reconstructed line. A significant disadvantage of using this architecture is that each bin of the accumulator consumes an additional bit of memory. Additionally, the author acknowledges that it cannot detect lines near the corners of an image, which is caused by on-chip memory limitations.

An investigation into the use of Vivado HLS as a development tool for LHT architectures is described by Solod *et al.* [90]. The authors designed a software model of the LHT using the C++ programming language and then used Vivado HLS to generate an LHT architecture design. The target image resolution was 1920×1080 pixels and the LHT architecture only consumed 77.5 BRAM tiles. The HPS was configured for $N_\rho = 2048$ and $N_\theta = 41$. The authors deliberately reduced the size of the HPS to use fewer on-chip memory resources.

3.3.3 Discussion

Many of the software implementations of the LHT discussed in Section 3.3.1 reduce the algorithm's computational complexity. For example, the Gradient LHT, FIHT2, PHT, PPHT, and KHT reduce the number of votes applied to the HPS, decreasing the number of arithmetic operations. However, very few LHT algorithms optimise the memory requirements of the HPS. The AHT is one example that optimises memory to implement line detection in early CPUs, which lacked the memory capabilities to process large image resolutions. However, the AHT requires random memory access, which is challenging to implement on an FPGA unless the candidate image is accessible using on-chip memory. Alternatively, the authors in [23] describe a memory-efficient LHT that applies votes to a compressed HPS. This work uses lossy compression of the HPS to reduce memory requirements. This thesis presents the ARLHT, which is a novel memory-efficient algorithm described in Chapter 6 and based on a modified version of the memory-compressed LHT. The ARLHT algorithm improves the memory allocation of the HPS and reduces the overall FPGA resource consumption compared to previously published LHT architecture designs.

Several FPGA implementations of the LHT in previously published works (see Section 3.3.2) deliberately reduce the size of the HPS or shorten the wordlength required to store votes in memory correctly. For instance, the authors in [89] and [21] use these design techniques to reduce the memory consumption of the HPS. These techniques constrain the operation of the LHT as the architecture in [21] cannot detect lines near the corners of an image, and the architecture presented in [89] may detect spurious lines. Other FPGA architectures of the LHT, such as the designs given in [13] and [88], restrict the discretisation step of the HPS along the θ-axis. This design technique may improve the resource efficiency of the LHT implementation. However, the architecture will only be suitable for bespoke applications as the accuracy of line detection is limited to specific angles. Zhou et al. [87] describe a parallel implementation of the LHT, which will be named the parallel LHT for the remainder of this thesis. The parallel LHT does not constrain the HPS to improve FPGA memory consumption and does not introduce any of the limitations discussed above. Furthermore, it is scaleable and parameterisable where the architecture can process any image resolution, and several design parameters can be customised, including δ_θ, δ_ρ, N_θ, and N_ρ. An implementation of the parallel LHT architecture is used later in Chapter 4 to test the HEP.

The parallel LHT architecture can be optimised to improve memory consumption and reduce multiplication requirements in FPGA devices. This optimisation is described in Chapter 5 of this thesis, where a novel memory bit-packing scheme is employed to reduce BRAM tile requirements and spatial domain symmetry is exploited to decrease the allocation of DSP slices. Note that these optimisations do not affect the accuracy of the LHT or introduce any significant limitations of the algorithm.

3.3.4 Key Findings

In summary, the list below presents the key findings of this literature review on software algorithms and FPGA architectures of the LHT.

- There are few investigations into the memory requirements of the HPS. Many reported software algorithms of the LHT ignore this issue and only consider improving its computational performance.

- It is possible to improve the memory consumption of the HPS algorithmically. For example, the memory-compressed LHT described in [23] can reduce memory requirements by 50%. Alternatively, the authors of the AHT in [12] have demonstrated memory savings that are 3000 times more efficient than the standard LHT. Notably, the memory efficiency of the memory-compressed LHT can be improved and is suitable for FPGA implementation, which is a novel contribution of this thesis and is detailed later in Chapter 6.

- Many reported FPGA architectures of the LHT require significant memory resources to store the HPS. For instance, the LHT architecture reported in [20] requires 180 36 Kb BRAM tiles for an image resolution of 333×333 pixels, which consumes 58% of the available BRAMs on the XCZ7UEV device. Researchers often constrain the size of the HPS, which introduces algorithm limitations. These limitations include LHT architectures that cannot detect lines near the corners of an image and may also detect spurious lines.

- There are currently no LHT architecture designs that exploit bit-packing schemes to reduce the memory requirements of the HPS. Additionally, there are no reported architectures that use spatial domain symmetry to reduce the computational complexity of the LHT algorithm. Chapter 5 presents a novel FPGA architecture that uses optimisations to implement the LHT efficiently.

- Each of the authors in this review do not describe how they validated the correct operation of their LHT architecture. In Chapter 4, a novel evaluation platform is presented that validates LHT architectures on the physical target device.

The key findings identified in this literature review were used to motivate the original research presented throughout this thesis.

3.4 Embedded Applications

The LHT is an essential tool for many embedded systems, including UAVs, industrial inspection solutions, and wireless communication technologies. This section explores

many of these embedded systems and several candidate applications that can leverage an FPGA to accelerate the LHT. The key requirements of the LHT for each application are discussed in Section 3.4.5, which motivates the original research presented in Chapters 4, 5, and 6 of this thesis.

3.4.1 Advanced Driver Assistance Systems

Vehicles require vision systems that can robustly detect lane markings to determine whether the vehicle has drifted out of its lane. Consider the illustration shown in Figure 3.20, which presents an aerial view of a vehicle equipped with a video camera to monitor the lane markings on each side of the vehicle.

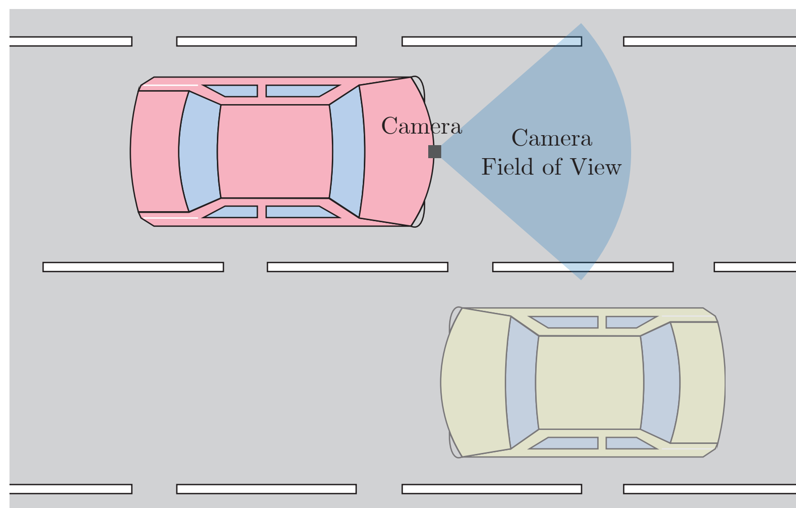


Figure 3.20: An illustration of a vehicle equipped with a video camera to monitor the lane markings on each side of the vehicle.

The vehicle's motion as it travels along the road can make lane detection challenging, as the lane markings will change position from one video frame to the next. Also, lane markings often contain gaps and are commonly obfuscated by objects such as other vehicles or pedestrians. A valuable feature of the LHT is that it can determine the exact position of a line in an image irrespective of its orientation or location. This capability of the LHT allows it to detect lines that change position between consecutive video frames. Additionally, the LHT is robust to partially occluded lines, allowing it to detect lines that have been partly obfuscated from the camera's view.

Figure 3.21a contains a digital image of road markings that will be processed using the LHT. Detected lines are reconstructed using (3.6) and (3.7) and are overlaid on top of the image as shown in Figure 3.21b for inspection. Notice that the LHT has detected the lane markings even though they contain significant gaps.

(a) A digital image of road markings.

(b) Detected lines overlaid on top of the image.

Figure 3.21: A digital image of road markings to be processed using the LHT (a). The detected lines are reconstructed and overlaid on top of the image (b).

MathWorks have published an example in [91], which uses the LHT to detect road markings and determine whether a vehicle has departed its lane. The example operates by first extracting road markings from a video frame and comparing these to the road markings in the subsequent frame. The system labels the lanes at each side of the vehicle and issues warnings when the vehicle crosses over the lane's boundaries. This system cannot target an FPGA, as it is not compatible with MathWorks *HDL Coder*.

Lane departure warning systems can be implemented using a combination of a Deep Neural Network (DNN) and the LHT to extract road markings. Chao *et al.* [5] describe a multi-lane detection system for Advanced Driver Assistance Systems (ADAS) applications. This system uses a trained DNN to detect binary lane pixels from input video frames. The LHT is applied to the resulting image and acts as a post-processing operation to determine a linear equation, which can be used to describe the detected lanes mathematically.

DNN based approaches for lane detection can be problematic for embedded systems as they are highly computational and energy inefficient. Traditional techniques can be less computationally demanding. Hajjouji *et al.* [22] developed a resource-efficient FPGA architecture of the LHT that detects road markings for lane detection. The authors recognise that lane markings are not horizontal or vertical when a camera is located at the front of a vehicle. Due to perspective, the lane markings appear to be diagonal to the camera, which allows the accumulator memory to be optimised by reducing the range of θ to specific angles.

3.4.2 Unmanned Aerial Vehicles

UAVs are often programmed to have autonomous flight routines that an operator does not control. This capability makes them very useful for automated inspection of agriculture, buildings, and structures. The illustration given in Figure 3.22 presents a selection of candidate application areas for UAVs.

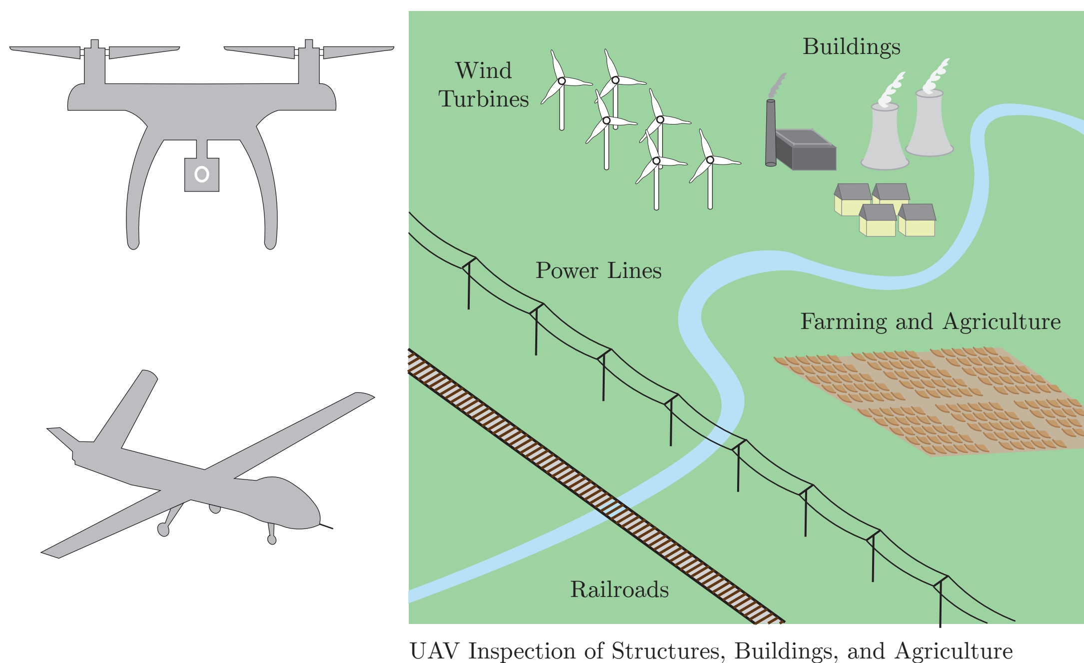
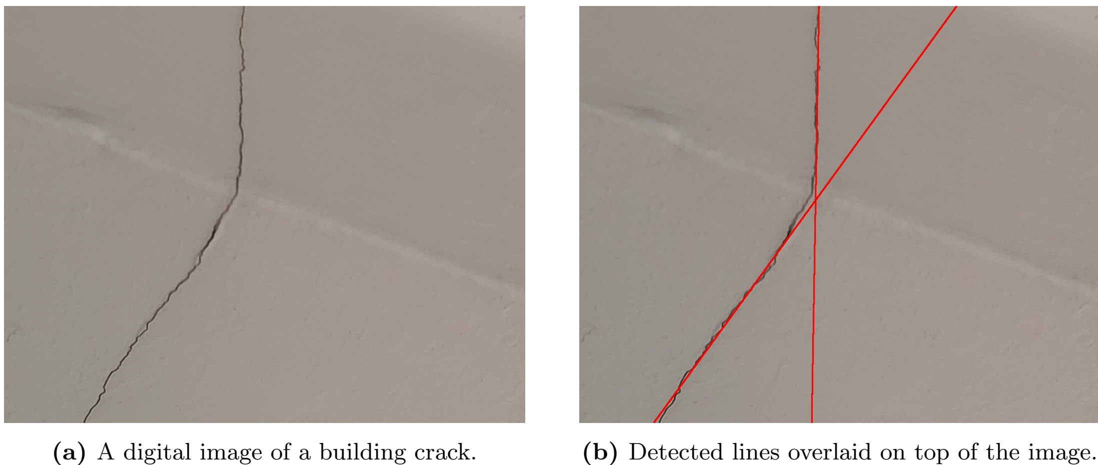


Figure 3.22: An illustration of a UAV equipped with a digital video camera that can perform automated tasks in several application areas, including power line analysis, crop line analysis, wind turbine tracking, and railroad evaluation.

UAVs can be used to assess the status of power lines, as described in [6]. The authors report that their proposed system can accurately extract power lines using the LHT from their dataset that was recorded on a UAV. Alternatively, UAVs can be used to evaluate railroad infrastructure. For instance, the authors in [7] successfully detect rail tracks in digital images taken from high-altitude drones. Their proposed system uses the LHT to detect lines in digital images, which are then labelled using a custom clustering algorithm. UAVs have also been used to inspect wind turbines. For example, the work in [92] uses the LHT to detect primary components of the wind turbine to support the UAV's navigation system as it performs its inspection. Lastly, UAVs can inspect structures such as buildings and bridges for cracks [93]. Figure 3.23 presents a simple example of using the LHT to detect straight cracks on a wall.



(a) A digital image of a building crack.

(b) Detected lines overlaid on top of the image.

Figure 3.23: A digital image of a building crack to be processed using the LHT (a). The detected lines are reconstructed and overlaid on top of the image (b).

The farming and agriculture industries also use UAVs to inspect fields and monitor crops. In [94], the authors describe an algorithm that can detect crop lines of a mango tree plantation from digital images taken by a drone. The LHT is used to identify crop lines in digital images of the plantation, which helps to address problems associated with fertilising and planting mango trees. Another interesting application of UAVs is given in [95], where the authors describe a technique to monitor weed growth amongst crops. In this application, the LHT is used to identify rows of crops, which are then combined with a CNN to detect weeds.

3.4.3 Industrial Inspection

The LHT is often required for industrial inspection applications to extract a Region of Interest (ROI), or to identify product defects such as cracks. An illustration depicting an industrial vision system is given in Figure 3.24.

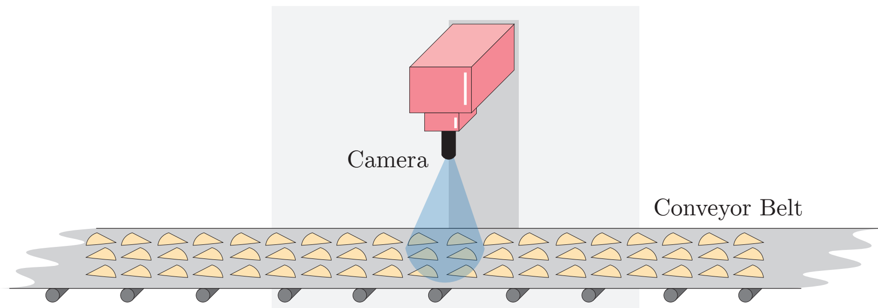
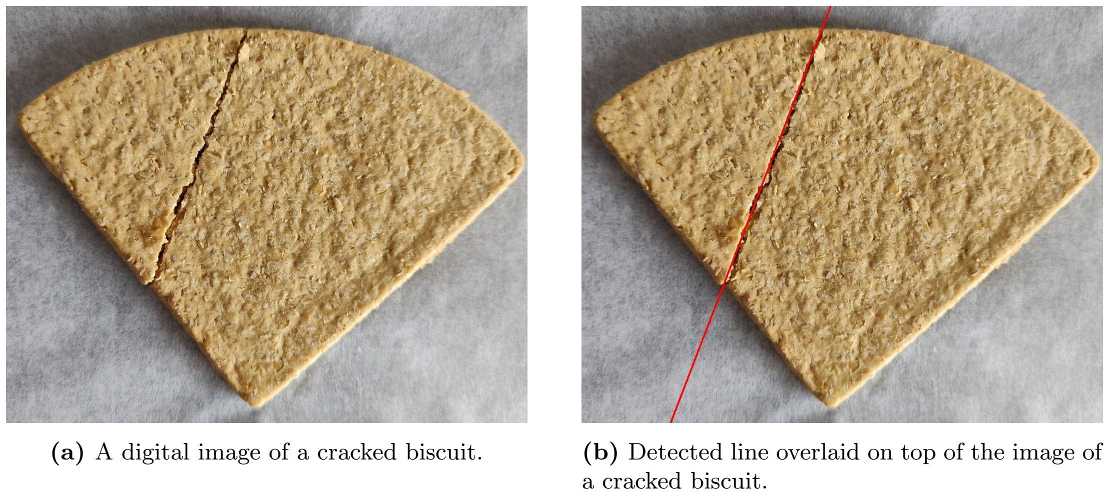


Figure 3.24: An illustration of an industrial vision system to inspect products. This particular diagram depicts a camera directed at a conveyor belt of food products.

Quality control is an important process during the manufacture of a product, as it identifies issues that are unacceptable for the consumer. However, quality control is time consuming and can be a limiting factor on the total factory yield. Industrial inspection systems often use cameras, as depicted in Figure 3.24, to accelerate quality control and detect product defects quickly. FPGAs can play an important role in interfacing to high performance cameras that operate using high resolutions and frame rates. The cameras are usually directed towards a product manufacturing line or conveyor belt and photograph or record the product as it moves past the camera. Digital images of the product must be processed quickly to identify defects, so that the product can be removed from the manufacturing line if needed.

The LHT has been used for industrial inspection systems. The authors in [96] describe an algorithm that uses the LHT to detect cracks in biscuits on a manufacturing line. This quality control process is important to ensure consumer satisfaction and to identify manufacturing issues quickly. Figure 3.25a contains an example of a biscuit that contains a crack. The LHT is used to detect lines in the image, which are then reconstructed for inspection. As shown in Figure 3.25b, the LHT has successfully detected a crack in the biscuit as indicated by the line drawn over the crack.



(a) A digital image of a cracked biscuit.

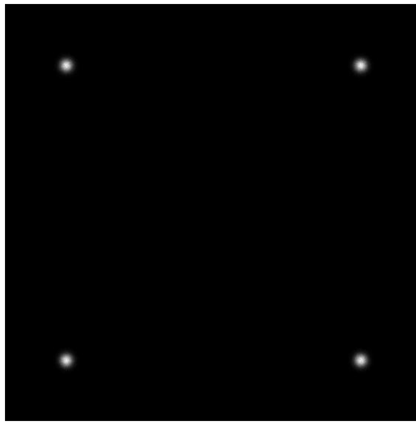
(b) Detected line overlaid on top of the image of a cracked biscuit.

Figure 3.25: A digital image of a cracked biscuit to be processed by the LHT (a). The detected line is reconstructed and overlaid on top of the biscuit image (b). Note that the line thickness has been increased in size for inspection purposes.

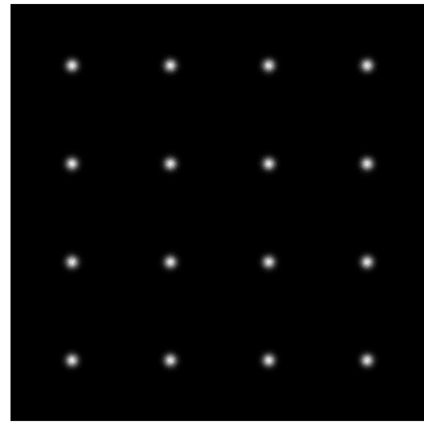
Wang *et al.* [97] also describe a defect inspection system that is used for evaluating plastic bottles. This system requires the LHT to extract an ROI from a digital image taken on a manufacturing line. The ROI contains the plastic bottle object, which is then input into a CNN that has been trained to identify bottle defects. The LHT in this scenario is primarily used as a preprocessing step, before the ROI is used in the CNN. The authors highlight that ROI extraction is important, as it reduces the time taken to process the digital image using the CNN.

3.4.4 Wireless Communications

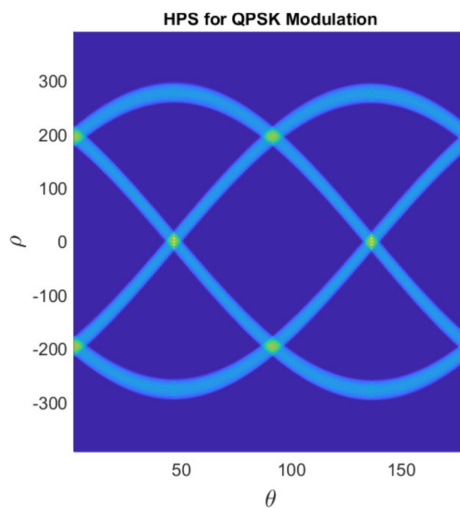
The LHT is emerging as a valuable tool in wireless communication systems. Two major applications involve classifying modulation schemes and detecting chirps in spectrogram plots. For example, Modulation Format Recognition (MFR) is a process in OWCs that identifies the modulation format adopted by a transmission system. An OWC receiver will typically perform MFR before demodulating a visible light signal to extract data. Figure 3.26 presents an example of applying the LHT to two digital modulation schemes, which are Quadrature Phase Shift Keying (QPSK) and a type of Quadrature Amplitude Modulation (QAM) with 16 symbols, known as 16-QAM. In each example, the HPS is plotted for inspection.



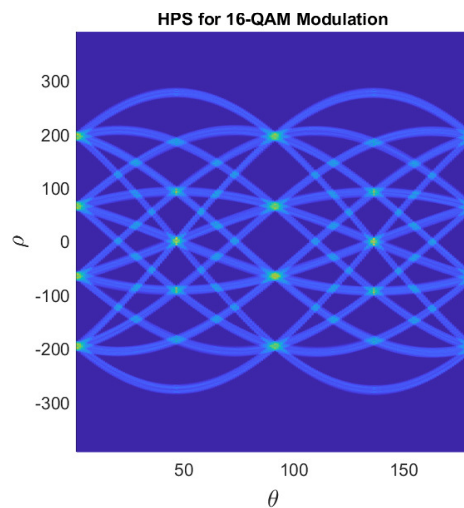
(a) Constellation of a QPSK modulation scheme.



(b) Constellation of a 16-QAM scheme.



(c) HPS of the QPSK constellation.



(d) HPS of the QAM-16 constellation.

Figure 3.26: The LHT is applied to an image of a QPSK constellation (a) and 16-QAM constellation (b), resulting in an HPS for each example given in (c) and (d), respectively.

MFR can be a challenging task when there is a significant level of channel noise that obfuscates the line of sight between the transmitter and receiver. Mohamed *et al.* [8] describe the use of the LHT to assist in applying MFR for OWC receivers. Their novel MFR scheme plots modulated OWC signals on a constellation diagram. The constellation is then edge detected and processed using the LHT to compute collinearity between received symbols. The output HPS is of particular importance as this initially trains a DNN to recognise the modulation scheme employed by the OWC signal. After training, the DNN processes the HPS of modulated OWC signals.

Another application of the LHT in wireless communications is to identify chirps in spectrogram plots. LFMCWs are commonly used in radar systems to determine the distance and velocity of objects approaching the radar antenna. LFMCWs increase in frequency over time, which is characteristic of a chirp signal. In spectrogram plots, LFMCWs appear as a steep line, which can be detected using the LHT. The authors in [9] describe an FPGA based system that can detect LFMCWs in low SNR environments. Their system deploys the LHT to detect collinear features in spectrogram plots, which are then used to characterise and describe LFMCWs in the environment.

3.4.5 Discussion of Requirements

This section explores the technical requirements of the embedded applications described in previous sections, such as the system latency and the configuration of the Hough parameters. These requirements motivate the design choices for the FPGA architectures of the LHT presented in Chapters 5 and 6 of this thesis.

Operational Latency

Vehicles and UAVs often require low-latency processing to perform safety-critical tasks. For example, a vehicle's vision system uses the LHT to determine if the vehicle has drifted out of its lane. This task must be completed within a guaranteed time to allow the driver to intervene. UAVs also have strict line detection requirements depending on the underlying application. For instance, a UAV can use the LHT to detect the main parts of a wind turbine to support its navigation system. The UAV may unintentionally strike the wind turbine if its navigation system does not receive the correct information from the vision system at the expected time.

Industrial inspection systems also require low-latency processing to operate efficiently. For instance, manufacturing vision systems detect lines in digital images of products to effectively inspect defects and quickly remove offending products from the manufacturing line. Reducing the latency of the quality inspection system can improve production output and increase the speed of the conveyor belt moving products between manufacturing stages.

In wireless communications, MFR systems must be able to quickly determine the modulation format of an OWC signal to optimise the efficiency of the data communications link. FPGA platforms can achieve low latency line detection in the above applications by accelerating the LHT algorithm.

Deterministic Processing

Embedded applications often require deterministic line detection, which refers to detecting lines in a digital image within a predictable or known amount of time. Many applications require deterministic processing, such as lane detection and industrial product inspection. The authors in [89] report a resource-efficient FPGA architecture of the LHT by storing the HPS in external memory. This technique can be problematic if other resources or hardware accelerators require external memory, as they will each compete for access. This architecture is non-deterministic because it applies the LHT on a best-effort basis. The FPGA architectures of the LHT in this thesis will use dedicated on-chip memories to store the HPS to accommodate applications that require deterministic line detection.

Angles of Operation

The authors in [22, 90] describe FPGA implementations of the LHT where they have reduced the operational range of θ to optimise on-chip memory consumption. While this design choice is an appropriate solution to reduce the memory requirements of the HPS, it restricts the LHT architecture for use in specific bespoke applications that can only detect lines over a reduced range of orientations. The FPGA architectures of the LHT in this thesis aim to target all applications while reducing the total on-chip memory consumption of the HPS. These LHT architectures are suitable for applications that require line detection across orientations of θ in the range $[0°, 180°)$.

Video Processing Standard

Several embedded applications will likely use a video camera or photo capture device that employs a media standard. The LHT architectures in this thesis aim to achieve the

FHD standard, which has an image resolution of 1920×1080 pixels and displays 60 fps. As presented in Chapter 4, an LHT architecture that achieves the FHD standard is expensive in terms of on-chip FPGA memory consumption. Memory can be optimised using bit-packing and memory compression techniques, as described in Chapters 5 and 6 of this thesis, respectively.

3.4.6 Key Findings

Many of the embedded applications described in Section 3.4 require an FPGA implementation of the LHT to achieve low operational latency and deterministic processing. Large FPGA devices (consisting of hundreds of thousands of logic elements) offer the necessary on-chip memory resources to store the HPS [19]. However, large FPGAs are financially expensive and have physically large package sizes that consume a significant amount of area on a Printed Circuit Board (PCB) and increase the overall weight of the final solution. The advantages of optimising the memory efficiency of the LHT on FPGA devices for embedded applications are listed as follows.

1. *Package Area* — Reducing the memory consumption of the HPS can increase the number of small FPGA devices that the LHT can target. For instance, a large Kintex UltraScale+ device named XCKU5P, has over 400 BRAM tiles for storing the HPS [19]. The dimensions of the package available for this device is 23×23 mm. However, a small FPGA, such as the Artix UltraScale+ device named the AU15P, can be purchased using a package that has dimensions of 11.5 $\times$ 9.5 mm [18]. The AU15P only contains 144 BRAMs, meaning it is unable to store the HPS of a parallel LHT for image resolutions such as 1280×720 pixels, when $\delta_\theta = 1^\circ$ and $N_\theta = 180$ (see Section 4.4.1 for parallel LHT implementation results). It is necessary to optimise the memory allocation of the HPS to target small FPGAs with fewer BRAM tiles and improve the package area on a PCB. This factor affects applications where the PCB area is a critical design parameter, such as motherboards for a UAV or road vehicle.

2. *Device Cost* — Large FPGA devices containing significant amounts of BRAM tiles (250 or higher) are financially expensive. Reducing the size of the FPGA required by an application can improve the overall project cost, as small FPGAs are cheaper to purchase. Overall, this factor significantly affects the project cost during mass production, where tens of thousands of FPGA units are purchased and assembled with other components on a manufacturing line. The size of the selected FPGA significantly affects the manufacturing costs for mass-produced technologies, including UAVs, vehicles, production line apparatus, and wireless communication equipment.

3. *Energy Efficiency* — Many embedded applications that use a battery source as their primary power supply will require energy-efficient solutions. For example, vehicles and UAVs each operate on batteries (although the former is likely to use a petrol or diesel engine). Portable spectrum monitoring solutions that require the LHT to detect chirps in spectrograms will also use batteries. FPGAs offer an energy-efficient solution to accelerating the LHT. It is necessary to optimise LHT architecture designs to use the least amount of logic elements possible on an FPGA, as this keeps energy consumption low. In particular, reducing BRAM tile and DSP slice consumption can decrease energy requirements.

4. *Reprogrammable* — An FPGA can be reprogrammed after it has been deployed in a product such as a vehicle or UAV. Suppose an essential algorithm, such as the LHT, consumes a sizeable area of the FPGA logic fabric. In that case, it can be challenging for developers to update the logic fabric with new hardware accelerators that feature new content. Optimising the resource efficiency of the LHT architecture design can be a valuable way of freeing FPGA logic fabric resources for other operations and tasks.

The key findings of this application review for the LHT listed above motivate the original research in Chapters 5 and 6 of this thesis.

3.5 Conclusion

The beginning of this chapter introduced the LHT by describing its inception using the slope-intercept representation of a straight line as a candidate template. The normal representation of a straight line was described as a robust template that could detect all lines in a digital image regardless of orientation. The Glide Reflection was then discussed, demonstrating that the HPS contained redundancy that could be removed to optimise memory allocation. The memory requirements of the HPS were then explored in detail, and it was found that placing the image origin in the centre of the image considerably reduces the size of the HPS, further decreasing memory requirements. An expression that computes the total memory consumption of the HPS (in bits) was also derived. An example of the LHT was then presented, and the corresponding HPS was investigated. Line reconstruction was also explored briefly.

A review of previously published literature relevant to the work in this thesis was presented. Several software implementations of the LHT were investigated, including the Gradient LHT and the memory-compressed Hough Transform, which are relevant to the work presented in Chapter 6. Several FPGA implementations of the LHT were explored, including multiplierless and resource-efficient architecture designs. Finally, several candidate applications that can leverage an FPGA architecture of the LHT were explored, including lane departure warning systems for vehicles and UAV inspection of powerlines and railroads. It was found that the device cost, package area, energy efficiency, and reprogrammable capabilities of FPGA solutions are important factors for embedded applications and motivate the design of a memory-efficient LHT architecture for FPGA implementation in Chapters 5 and 6.

The next chapter details a novel evaluation platform for designing and validating FPGA architectures of the LHT. An LHT architecture design will be introduced and analysed for timing performance and FPGA resource consumption. In particular, the memory requirements of the HPS will be investigated.

Chapter 4

The Hough Evaluation Platform

4.1 Introduction

Special development environments and design tools are typically required to perform FPGA architecture validation and analysis. These environments enable rapid FPGA prototyping and iterative development of architecture designs. MathWorks *HDL Coder* reference designs enable rapid FPGA prototyping, while the PYNQ framework allows users to deploy and execute Jupyter Notebooks directly on their Zynq MPSoC platforms. Both environments offer reproducible embedded systems and repeatable results. For this reason, it is useful to combine these tools to improve the analysis of FPGA architectures and enhance the dissemination of research findings.

The primary contribution of this chapter is a novel evaluation platform for FPGA architectures of the LHT, named the Hough Evaluation Platform (HEP), which was first published in [27]. An FPGA evaluation platform enables researchers and developers to rapidly prototype and evaluate FPGA-based applications as the evaluation platform readily provides constraints, interconnects, interfaces, and a variety of other preconfigured components. In particular, the novel HEP enables the rapid development of LHT architectures with MathWorks *HDL Coder* and supports the hardware testing and validation on the physical target device using the PYNQ framework. Upon applying an image to a custom LHT architecture, the HEP can be used to accurately measure the processing time and acquire the output HPS for visualisation and analysis. To the

author's knowledge, this is the first FPGA development and evaluation environment that combines PYNQ and MathWorks *HDL Coder*. The HEP is used later in Chapters 5 and 6 for the development and validation of novel LHT architecture designs.

4.2 Evaluation Platform Design

In this section, the novel evaluation platform for rapidly designing and deploying LHT architectures on Zynq MPSoC devices is described. The underlying architecture of the HEP will be detailed, and its primary features will be explored. These features include unique plotting and inspection capabilities for the HPS and accurate processing time measurements. A system overview of the HEP can be seen in Figure 4.1.

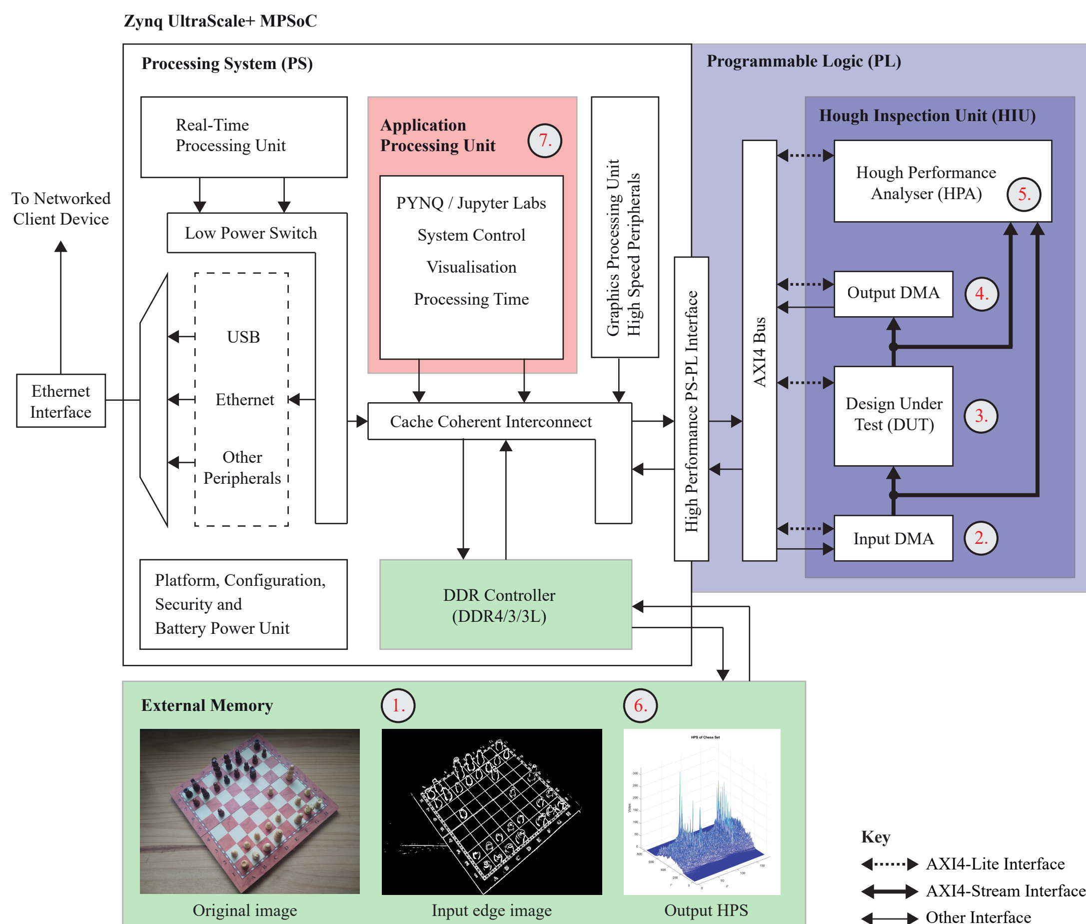


Figure 4.1: System overview of the HEP that targets the Zynq MPSoC. The PL accelerates the user's custom LHT architecture, while the PS plots and evaluates the resulting HPS.

The HEP is implemented entirely on the Zynq MPSoC device. The PL consists of the Hough Inspection Unit (HIU), which contains a measurement module named the Hough Performance Analyser (HPA). The PS is host to the PYNQ software framework. The HIU and PYNQ framework can access shared off-chip memory and communicate using the AXI bus. Figure 4.1 contains several points of interest, numbered 1 to 7, which outline the operation of the HEP. These points are described below.

1. The user selects a candidate test image and loads it into external memory. Depending on architecture requirements, the user may pre-process the image in software to extract edges.

2. The candidate test image is retrieved from external memory by the input AXI DMA in the HIU. While the image is transferred from external memory into the PL, the HPA begins measuring the architecture's processing time.

3. Image processing operations are applied to the candidate test image by the Design Under Test (DUT). The DUT contains the user's custom LHT architecture, which may contain edge detection stages or a custom image processing pipeline.

4. When DUT processing is complete, the result is sent to the output AXI DMA to be written into external memory. The output of the DUT contains the resulting HPS of the candidate test image.

5. While the output of the DUT is transferred to the AXI DMA, the HPA monitors the DUT for a signal that indicates it has successfully completed its operation.

6. Once the output DMA transfers the HPS into external memory, it can then be accessed by the PYNQ framework and analysis software.

7. Existing Python libraries that are available through the PYNQ framework, such as Plotly [98] and NumPy [99], are used to evaluate and plot the HPS. The user may save hardware test results and perform further experiments to compare LHT architectures and corresponding Hough parameters. The HPS can be easily plotted and inspected.

To reduce the complexity of operation, the HEP was designed to be completely autonomous and accessible using a simple browser based interface. The remainder of this section will detail the design of the HEP and present the hardware architectures of the HIU and HPA. The HEP's analysis capabilities and software design will also be presented.

4.2.1 The Hough Inspection Unit

The purpose of the HIU is to interface the user's custom LHT architecture to the Zynq MPSoC's PS, via the external memory. Additionally, the HIU should also measure the processing time of the user's architecture using the HPA module. The HIU is implemented entirely in the Zynq MPSoC's PL and contains an AXI DMA, the HPA, and the user's custom architecture. A detailed diagram illustrating the HIU is presented in Figure 4.2. Note that the AXI4-Lite interfaces are connected to the same AXI bus. Although not specifically shown in Figure 4.2, each component in the HIU shares the same clock and reset signals to reduce design complexity.

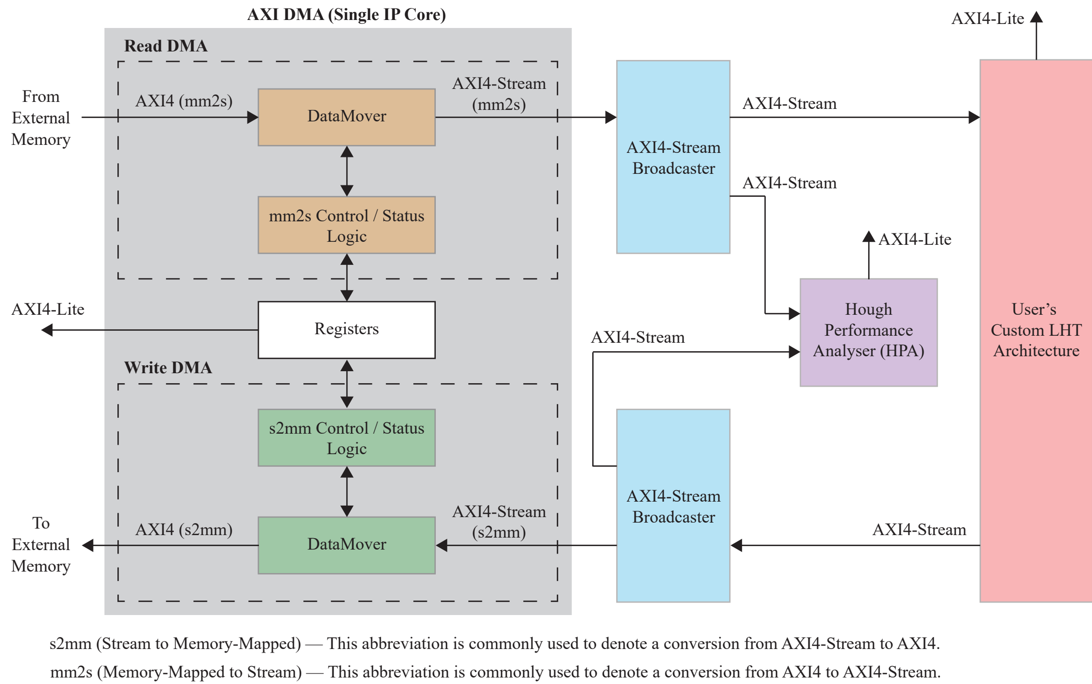


Figure 4.2: Detailed diagram of the HIU, consisting of an AXI DMA, the HPA, AXI4-Stream Broadcasters, and the user's custom architecture.

The interfaces used between each component in the HIU are AXI4, AXI4-Stream, and AXI4-Lite. The AXI DMA uses AXI4 to transfer fixed bursts of data between the PL and external memory. AXI4-Lite is used by the AXI DMA, the HPA, and the user's custom LHT architecture to communicate with control and status registers. For point-to-point data streaming, the AXI4-Stream interface enables a direct flow of data between the AXI DMA and the custom LHT architecture. There are two IP Cores in the HIU that are provided by AMD and distributed through the Vivado Design Suite's IP Catalogue. These are the AXI DMA and AXI4-Stream Broadcaster cores. Each of these are briefly described below.

AXI DMA

Hardware accelerators in the Zynq MPSoC's PL are able to communicate with off-chip memory using the AXI DMA IP Core provided by AMD. As illustrated in Figure 4.2, there are two data movers inside the AXI DMA IP Core. The Read DMA allows data to be fetched from external memory and transferred into the PL. Similarly, the Write DMA allows data to be transferred from the PL and written to external memory. Each read and write operation uses the AXI4 interface to communicate with the Zynq MPSoC's DDR controller, previously shown in Figure 4.1. The PL communicates with the Read and Write DMAs using the AXI4-Stream interface.

AXI4-Stream Broadcaster

The AXI4-Stream Broadcaster IP Core provides a simple way of duplicating an AXI4-Stream Slave interface into two or more AXI4-Stream Master interfaces. This functionality is often required when connecting an AXI4-Stream interface to more than one IP Core in an FPGA system. The HIU requires an AXI4-Stream Broadcaster IP Core to duplicate the AXI4-Stream interface from the read DMA. One of the AXI4-Stream interfaces is routed to the user's custom LHT architecture, and the other interface is routed to the HPA for performance analysis. Another AXI4-Stream Broadcaster is required at the output of the user's custom LHT architecture to duplicate the resulting AXI4-Stream interface and route it to the Write DMA and HPA.

4.2.2 The Hough Performance Analyser

The purpose of the HPA is to measure the processing time of the user's custom LHT architecture on the physical target device. In previously published works (see Section 3.3), architecture processing time is measured using theoretical analysis, simulations, or a combination of both approaches. Furthermore, many authors only state the processing speed of their proposed architectures and do not specify how these measurements are performed. The HPA assesses the processing time of custom LHT architectures while they operate on the physical target device. Measuring the processing time of an architecture in this way is more effective than theoretical calculations and simulation results. This section describes the architecture of the HPA.

Design Specification

The majority of the HPA's functionality is implemented in the PL of the Zynq MPSoC device. Therefore, the HPA can leverage the deterministic performance and parallel processing capabilities of an FPGA to measure the processing time of the user's custom architecture. The design specification of the HPA is listed below, where architecture constraints and limitations are introduced to simplify design and implementation. The HPA should be able to achieve the following.

1. Communicate with the AXI4-Stream Slave interface of the read DMA and the AXI4-Stream Master interface of the user's custom architecture.

2. Monitor the AXI4-Stream Slave interface of the read DMA for valid data.

3. Measure the architecture processing time up to a maximum of 5 seconds or more.

4. Achieve processing time accuracy equivalent to the period of the system clock that drives the user's custom architecture.

5. Monitor the output AXI4-Stream Master interface of the user's custom architecture for the end of processing signal (TLAST).

6. Report the processing time measurement using an AXI4-Lite register. The HPA should also be resettable using the AXI4-Lite interface.

Specifications 1, 2, 5, and 6 describe how the HPA should integrate into the HIU and communicate with neighbouring IP Cores. Specifications 3 and 4 introduce accuracy and measurement limitations to constrain FPGA resource consumption. The limitation introduced in Specification 3 was necessary to ensure an upper limit is set and prevent unnecessary allocation of FPGA resources. If the user requires more time, they will be able to modify the HPA design manually. Specification 4 is necessary to calculate the architecture processing time accurately. The custom architecture will be operating at a user-specified clock speed, which can be configured during embedded system integration described in Section 4.3. Since the system clock can be changed by the user between development builds, the HPA will need to be flexible to accommodate Specification 4. The processing time accuracy is set to the period of the system clock, as this is the minimum time step during architecture operation.

Architecture Development

The HPA architecture does not contain many components and can be readily created using an HDL counter, an AXI4-Lite interface, and two AXI4-Stream interfaces. An illustration of the HPA architecture design is presented in Figure 4.3. Notice that the configuration of the HDL counter is annotated on the diagram.

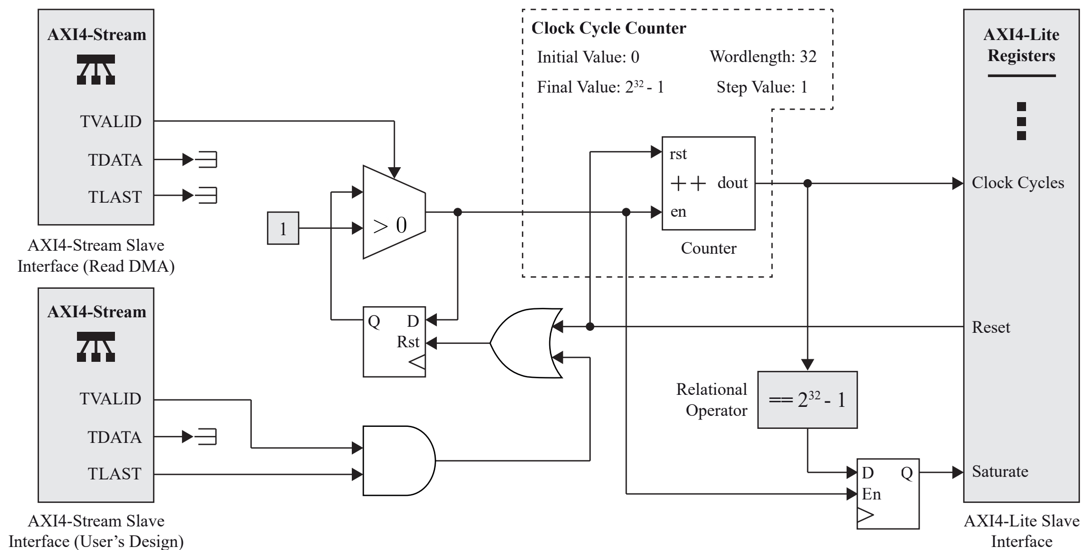


Figure 4.3: FPGA architecture of the HPA. The system contains two AXI4-Stream interfaces, three AXI4-Lite registers, control circuitry, an HDL counter, and a relational operator.

Two AXI4-Stream interfaces are shown on the left of Figure 4.3. The AXI4-Stream interface from the read DMA is used to transfer the test image to the HPA. The TVALID signal of this interface is required to determine the first valid data transfer of the input image. Similarly, the AXI4-Stream interface from the user's custom architecture is responsible for transferring the HPS to the HPA. The TLAST and TVALID signals are used to identify the last data transfer of this interface.

Three AXI4-Lite registers are shown to the right of Figure 4.3. The first register stores the HDL counter output and the second register can apply a reset signal to the HDL counter. The reset signal is asserted by writing to the register from software. The third register, named 'saturate', is used to detect when the counter has reached its maximum value. The saturate register informs the user that the HPA is unable to measure the processing time of the LHT architecture as the counter has saturated.

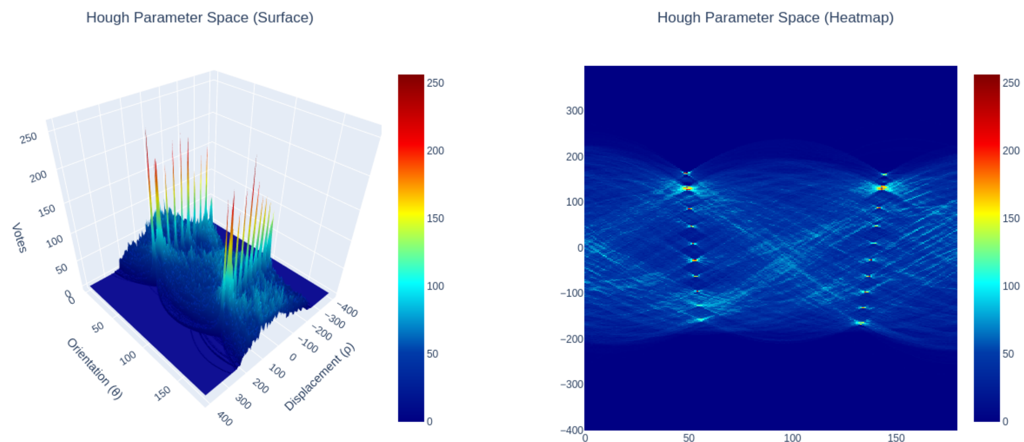
The HPA contains simple control circuitry that determines the first valid data transfer of the input image and the last data transfer of the corresponding HPS. The HDL counter calculates the number of clock cycles that have passed since the first valid data transfer was detected. When the last data transfer of the HPS is detected, the HDL counter stops accumulating. The user can calculate the processing time from software by reading the clock cycles register and dividing its value by the system clock frequency (see Section 4.2.3 for an example). The HDL counter uses a 32 bit wordlength and saturates when it reaches its final value. This counter is suitable for measuring 5 seconds of processing time for system clock frequencies less than, or equal to, 858 MHz.

4.2.3 Visualisation and Analysis

After the user's architecture has processed the test image, the Jupyter Lab environment can be used to visualise and inspect results. Jupyter can leverage existing plotting and analysis libraries, such as Matplotlib [100] and Plotly, to visualise the HPS. In this work, the Plotly library is used as it is simple to configure and very well supported with documentation. Jupyter is also used to calculate the processing time of the architecture using the clock cycles register in the HPA (shown to the right of Figure 4.3). This section details the HPS inspection and measurement capabilities of the HEP.

HPS Inspection

Jupyter Lab and Plotly can be used to visualise and interact with the HPS after it is retrieved from the user's LHT architecture. There are many plotting Application Programming Interfaces (APIs) that can be selected from the Plotly Graphics Objects library [98], which is typically written as `plotly.graph_objs`, or abbreviated as `go`. Figure 4.4, presents the HPS of the chessboard edge image (previously shown in Figure 3.7d on page 60) when plotted using the `go.surface` and `go.heatmap` APIs.



(a) HPS plotted using the `go.surface` API from the Plotly Graphics Objects library.

(b) HPS plotted using the `go.heatmap` API from the Plotly Graphics Objects library.

Figure 4.4: HPS visualisation using `go.surface` (a) and `go.heatmap` (b) interactive plotting APIs. The user can interact with the plots by zooming, panning, and hovering their mouse cursor over the plot to reveal cell data.

Each plot of the HPS is easily inspected in a web browser and can be saved and compared to results from other experiments. This testing environment is useful to ensure correct operation of the user's LHT architecture on hardware. The output HPS can be compared to a software model operating in the Zynq MPSoC's PS, which increases the effectiveness of the system for architecture validation. The user has the freedom to also use different plotting tools that are supported by Python, such as Matplotlib, Bokeh [101], Seaborn [102], and Ggplot [103]. Alternatively, the resulting HPS can be offloaded to MATLAB so that it may be inspected using MATLAB plotting tools [104].

Processing Time Calculation

The HPA module described in Section 4.2.2 allows the user to accurately measure the processing time of their custom LHT architecture. The HPA module uses a single 32 bit counter to maintain a running total of the number of clock-cycles that have passed since the architecture began operating. When the architecture has completed its operation, the clock-cycles AXI4-Lite register in the HPA module stores the total number of clock-cycles required by the architecture to process an image. The processing time can be easily calculated by multiplying the number of clock-cycles by the system clock period.

For example, a 640×480 pixel edge image is transferred into an LHT architecture that uses a system clock frequency of 150 MHz. The total number of clock-cycles reported by the clock-cycles register is 451,201. The total processing time is calculated as given in (4.1), and rounded to two decimal places.

$$\begin{aligned} \text{Processing Time} &= \frac{\text{Clock Cycles}}{\text{System Clock Frequency}} \\ &= \frac{451,201}{150\,\text{MHz}} \\ &= 3.01\,\text{ms} \end{aligned} \tag{4.1}$$

4.3 Embedded System Integration

This section aims to provide insight into the development of the HEP using MathWorks and the Vivado Design Suite. Several areas of the HEP's design using *HDL Coder* will be investigated. These include implementing the *HDL Coder* plug-in files, creating the IP Integrator block design, and developing automated device programming code. Additionally, embedded PYNQ software is also required to control and interface with the HEP for visualisation and analysis purposes. The available properties, methods, and classes of the HEP software drivers and APIs will be explored. Finally, a typical embedded system integration workflow using the HEP will be described and demonstrated.

4.3.1 The HDL Coder Reference Design

This section describes the integration of the HEP reference design with MATLAB and Simulink. The HEP reference design in this thesis follows the same design principles as outlined in the MathWorks documentation, which allows users to create their own custom *HDL Coder* reference designs [105]. The implementation of associated plug-in files, custom callback functions, and the general HEP workflow is detailed. Code snippets are investigated in this section when required. Note that the complete source code for the HEP can be found electronically in [106]. The versions of software packages used in this work are also important, as syntax and functionality may differ between releases. The software packages used in this work are MATLAB R2020a, the Vivado Design Suite 2020.1, and PYNQ v2.7 for the ZCU104 development board.

Reference Design Requirements

HDL Coder requires information about the underlying architecture and target platform to rapidly generate embedded system designs for the HEP. The following plug-in files are required to implement the HEP and are available electronically in [106].

- *board_design.tcl* — Contains the IP Integrator design that should be built during embedded system integration. *HDL Coder* does not parse this file for information. Instead, the file is passed to IP Integrator to build the required board design. The contents of *board_design.tcl* is described further in Section 4.3.2.

- *plugin_board.m* — Provides information about the target FPGA/SoC device, programming options, and the supported tool for bitstream generation. Additionally, information about FPGA pin constraints can be provided if required.

- *plugin_rd.m* — Provides *HDL Coder* with details of the underlying architecture design implemented in the *board_design.tcl* file. Details include system clocks and resets, the target IP Integrator board design file, AXI4 interface connections, IP Core repository locations, and custom callback functions to be executed during the HDL workflow.

- *hdlcoder_board_customization.m* — When *HDL Coder* is launched, all MATLAB paths are searched for files that are named *hdlcoder_board_customization.m*. This file provides *HDL Coder* with locations for active board registration files such as *plugin_board.m*.

- *hdlcoder_ref_design_customization.m* — When *HDL Coder* is launched, all MATLAB paths are searched for this file, as it provides *HDL Coder* with locations for active reference designs, such as *plugin_rd.m*.

Reference Design Plug-In Files

HDL Coder reference designs require board registration plug-in files to store information about the target platform. The development board used throughout this thesis is the ZCU104, which is host to the XCZU7EV-2E device. The board plug-in file, *plugin_board.m*, is configured to inform *HDL Coder* about the target development platform, as shown in Listing 4.1. The data presented is used to inform Vivado of the target device, so that it can prepare the implementation environment accordingly.

Listing 4.1: *plugin_board.m* — ZCU104 board plug-in registration file (lines 4-11).

```
4   % Construct board object
5   hB = hdlcoder.Board;
6   hB.BoardName     = 'ZCU104 Development Board';
7
8   % FPGA device information
9   hB.FPGAVendor    = 'Xilinx';
10  hB.FPGAFamily    = 'Zynq UltraScale+';
11  hB.FPGADevice    = 'xczu7ev-ffvc1156-2-e';
```

A particularly interesting file is *plugin_rd.m* as it contains information about the IP Integrator block design, clock and reset signals, and associated AXI4 interfaces. The reference design complexity can be reduced by limiting the user to AXI4-Stream interfaces at the input and output of their custom LHT architecture. Furthermore, AXI4-Stream interfaces are already used by the AXI DMA and HPA, further motivating this decision.

The AXI4-Stream interfaces should be constrained to 32 bit wordlengths, which will provide enough bits to interface 8 bit Red, Green, and Blue components. Finally, the AXI4-Stream interface should be connected to the AXI4-Stream Broadcaster IP Cores at the slave and master side of the user's architecture. This functionality is implemented in Listing 4.2.

Listing 4.2: *plugin_rd.m* — Add AXI4-Stream interfaces to the reference design (lines 53-61).

```
53  % add AXI-Stream interfaces
54  hRD.addAXI4StreamInterface( ...
55  'MasterChannelNumber', 1, ...
56  'SlaveChannelNumber', 1, ...
57  'MasterChannelConnection', 'axis_broadcaster_s2mm/S_AXIS', ...
58  'SlaveChannelConnection', 'axis_broadcaster_mm2s/M00_AXIS', ...
59  'MasterChannelDataWidth', 32, ...
60  'SlaveChannelDataWidth', 32, ...
61  'InterfaceID', 'AXI4-Stream');
```

In a similar way, the AXI4-Lite interface on the user's IP Core can be integrated into the system using *plugin_rd.m*. This functionality is demonstrated in Listing 4.3. Notice that the AXI4-Lite interface will be added to the master address space of the Zynq's PS and the base address is static i.e. 0xA0030000. The interface is also connected to an AXI interconnect IP Core named *ps8_0_axi_periph*, which is used to collate and manage AXI communications between the PS, and IP Cores in the PL.

Listing 4.3: *plugin_rd.m* — Add AXI4-Lite interface to the reference design (lines 47-51).

```
47  % add AXI4 and AXI4-Lite slave interfaces
48  hRD.addAXI4SlaveInterface( ...
49      'InterfaceConnection', 'ps8_0_axi_periph/M03_AXI', ...
50      'BaseAddress',         '0xA0030000', ...
51      'MasterAddressSpace',  'zynq_ultra_ps_e/Data');
```

The clock and reset signals are connected to the user's IP Core. This functionality is presented in Listing 4.4. The clock is sourced from a built-in Vivado IP Core named the clock wizard. The clock wizard is able to generate user specified clock frequencies that are passed from *HDL Coder* into the IP Integrator design. The default frequency of the clock wizard is 150 MHz, but it can be configured by the user to several frequencies between 10 MHz and 250 MHz.

Listing 4.4: *plugin_rd.m* — Add clock and reset signals to the reference design (lines 37-45).

```
37  % add clock interface
38  hRD.addClockInterface( ...
39      'ClockConnection',   'clk_wiz/clk_out1', ...
40      'ResetConnection',   'proc_sys_reset_clk_wiz/peripheral_aresetn',...
41      'DefaultFrequencyMHz', 150,...
42      'MinFrequencyMHz', 10,...
43      'MaxFrequencyMHz', 250,...
44      'ClockModuleInstance', 'clk_wiz',...
45      'ClockNumber', 1);
```

Lastly, the reference design will give the user the opportunity to enter the Internet Protocol (IP) address of the ZCU104 development board. The IP address will allow a custom programming routine to transfer the generated bitstream and driver software files to the file system on the ZCU104 platform. This functionality is implemented as shown in Listing 4.5.

Listing 4.5: *plugin_rd.m* — Add IP address parameter to reference design (lines 12-16).

```
12  % Add optional custom parameter for setting the ip address of the board.
13  hRD.addParameter( ...
14      'ParameterID',   'IPAddress', ...
15      'DisplayName',   'IP Address of ZCU104', ...
16      'DefaultValue',  '192.168.2.99');
```

Custom Programming Callback Function

During a typical *HDL Coder* workflow, the target FPGA device is programmed with the generated bitstream. Since the PYNQ framework controls bitstream programming, the programming step is modified to pass the bitstream and relevant drivers to PYNQ instead. This functionality is implemented in a custom programming callback function. The programming callback should create a new folder in the Jupyter workspace to store the driver software files and the bitstream. Secure Shell (SSH) is required to access PYNQ remotely. PuTTY is a software program that provides SSH functionality for Windows operating systems [107]. There are two commands provided by PuTTY that are useful for issuing SSH requests and transferring files from the local computer to the remote platform. These commands and their usage are listed as follows:

- `plink [options] [user@]host [command]`
- `pscp [options] source [source...] [user@]host:target`

The `plink` command allows the local computer to create a new folder in the target platform's file system. Once the folder is created, the `pscp` command is used to transfer the bitstream and driver files to the newly created folder. Putty is particularly useful, as the user's name and password can be passed into the `plink` and `pscp` commands as an optional parameter. The user does not need to interact with the custom programming process and the design files are seamlessly transferred between the local computer and remote platform.

Listing 4.6 presents lines 22 to 33 of the custom programming callback function. To begin, the IP address of the ZCU104 development platform is obtained from an internal *HDL Coder* object, known as the `infoStruct`. Then, the current date and time is used to create a unique character string. The unique string and IP address are then passed into an SSH command, which is issued to the ZCU104 platform using `plink`. The SSH command creates a new folder named after the unique character string that was formed from the date and time. The new folder is created in the Jupyter workspace and is accessible to the user.

Listing 4.6: *callback_CustomProgrammingMethod.m* — Obtain the IP address from the infoStruct object, obtain the date and time, and create a new folder in Jupyter using SSH and PuTTY (lines 22-33).

```
22  status = false;
23
24  % Set IP Address and open default browser
25  ipAddress = infoStruct.ParameterStruct.IPAddress;
26  web(['http://', ipAddress, ':9090/lab'])
27
28  % Set date and time and correct format for folder directory
29  dt = datetime;
30  dt.Format = 'ddMMyy_ssmmHH';
31
32  % Make Jupyter working directory
33  plinkCmd = "plink -ssh xilinx@" + string(ipAddress) + " -pw xilinx ";
```

A similar procedure is used to transfer the generated bitstream and driver software files into the newly created folder in the Jupyter workspace. This procedure is presented in Listing 4.7. Initially, the address location of the generated bitstream file is obtained. The `pscp` command is then used to transfer the bitstream into the remote Jupyter folder. This operation is repeated for the hardware hand-off file, which is a file generated by Vivado describing the contents of the IP Integrator block design. Similarly, the driver directory is also located, which contains the driver software files for operating the HEP (discussed further in Section 4.3.3). The contents of the directory are then parsed and the stored files are transferred to the remote folder using the `pscp` command.

Listing 4.7: *callback_CustomProgrammingMethod.m* — Transfer the generated bitstream and driver software files into the remote Juypter folder (lines 35-61).

```
35  system(plinkCmd + mkdirCmd);
36
37  % Transfer bitstream file
38  bitstreamDir = fullfile(pwd, infoStruct.ToolProjectFolder, ...
39      'vivado_prj.runs','impl_1','zcu104_hep_wrapper.bit');
40  pscpBitCmd = "pscp -pw xilinx " + bitstreamDir + " xilinx@" + ipAddress ...
41      +":/home/xilinx/jupyter_notebooks/hep/"+string(dt)+"/zcu104_hep.bit";
42  system(pscpBitCmd);
43
44  % Transfer hardware handoff file
45  hwhDir = fullfile(pwd, infoStruct.ToolProjectFolder, 'vivado_prj.srcs', ...
46      'sources_1', 'bd', 'zcu104_hep', 'hw_handoff', 'zcu104_hep.hwh');
47  pscpHwhCmd = "pscp -pw xilinx """ + hwhDir + """ xilinx@" + ipAddress ...
48      + ":/home/xilinx/jupyter_notebooks/hep/" + string(dt);
49  system(pscpHwhCmd);
50
51  % Transfer driver files and notebook
52  driverDir = replace(mfilename('fullpath'), ...
53      'callback_CustomProgrammingMethod', 'drivers\');
54  dirContents = dir(driverDir);
55  for idx = 1:length(dirContents)
56      file = dirContents(idx);
57      pscpDriverCmd = "pscp -pw xilinx """+driverDir+file.name + ...
58          """ xilinx@"+ipAddress+":/home/xilinx/jupyter_notebooks/hep/" ...
59          +string(dt);
60      if contains(file.name, '.py') || contains(file.name, '.ipynb') || ...
61          contains(file.name, '.jpg')
```

4.3.2 The IP Integrator Block Design

To support rapid prototyping, the HDL Coder reference design workflow creates an IP Integrator block design during hardware system integration. The block design is sourced from a pre-defined script using Tcl. Upon building the block design, the user's custom LHT architecture is inserted into the system at pre-determined locations specified in the HDL Coder reference design plug-in files. The block design file for the HEP can be found electronically in [106] for inspection. The same block design can be presented graphically, as shown in Figure 4.5. The clock, reset, and interrupt signals have been removed to simplify the diagram. Six points of interest have also been highlighted.

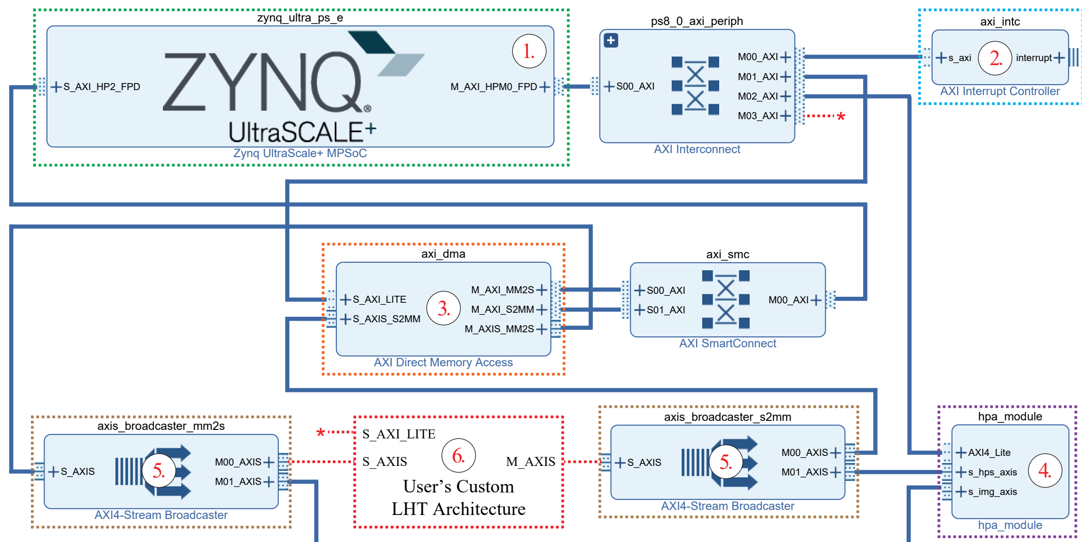


Figure 4.5: The IP Integrator block design for the HEP *HDL Coder* reference design.

At point 1 in Figure 4.5, the Zynq UltraScale+ MPSoC IP Core interfaces with the FPGA design using two AXI ports; these ports are named S_AXI_HP2_FPD and M_AXI_HPM0_FPD. The AXI master port supports AXI4-Lite single-beat communication between the FPGA design and the software operating in the PS. The AXI slave port is useful for transferring fixed-bursts of image data between off-chip memory and the AXI DMA. Each AXI port contains an AXI Interconnect or an AXI SmartConnect in their data path. These interconnect cores are provided by AMD and support communication between an AXI port and two or more AXI4 interfaces.

Point 2 and 3 in Figure 4.5 are the AXI Interrupt Controller and AXI DMA IP Cores, respectively. The interrupt controller is part of the PYNQ hardware design methodology and is required to handle interrupt assertions from the PL to the PS [108]. Although not shown in Figure 4.5, the DMA uses two interrupts to communicate events to software operating in the PS. These interrupts are connected between the AXI DMA and AXI Interrupt Controller. In particular, the interrupts communicate the successful completion of read and write transactions between the PL and PS, or report the occurrence of a system error. Formally, interrupts that communicate between the PL and the PS are known as Shared Peripheral Interrupts (SPIs).

Upon receiving image data, the AXI DMA transfers the data to the user's custom architecture. To facilitate processing time measurements, the AXI4-Stream is duplicated using AXI4-Stream Broadcaster IP Cores, as shown at point 4. The duplicated stream is sent to the HPA module, presented at point 5. The HPA can then measure the number of clock-cycles required to process the candidate test image and retrieve results. At point 6, the user's custom architecture will be inserted during embedded system integration using the *HDL Coder* workflow. Notice that the custom architecture consists of three AXI4 interfaces. These are the primary slave and master AXI4-Stream interfaces for communicating with the AXI DMA, and an AXI4-Lite slave interface connected to the M_AXI_HPM0_FPD port, via the AXI Interconnect.

4.3.3 Control and Analysis Software

The HEP requires software drivers to operate correctly. Python classes have been created for the PYNQ overlay and the HPA IP core. These are the *HoughEvaluation* and *HoughPerformance* classes, respectively. Each class inherits all the methods and properties from classes in the core PYNQ library [109]. A Unified Modeling Language (UML) diagram of the HEP software drivers is given in Figure 4.6. Several PYNQ library files are used by the HEP. The main files are *overlay.py* and *dma.py*, which contain the *Overlay* and *DMA* classes, respectively. The HEP also contains custom Python classes stored in two files: *hep.py* and *hpa.py*.

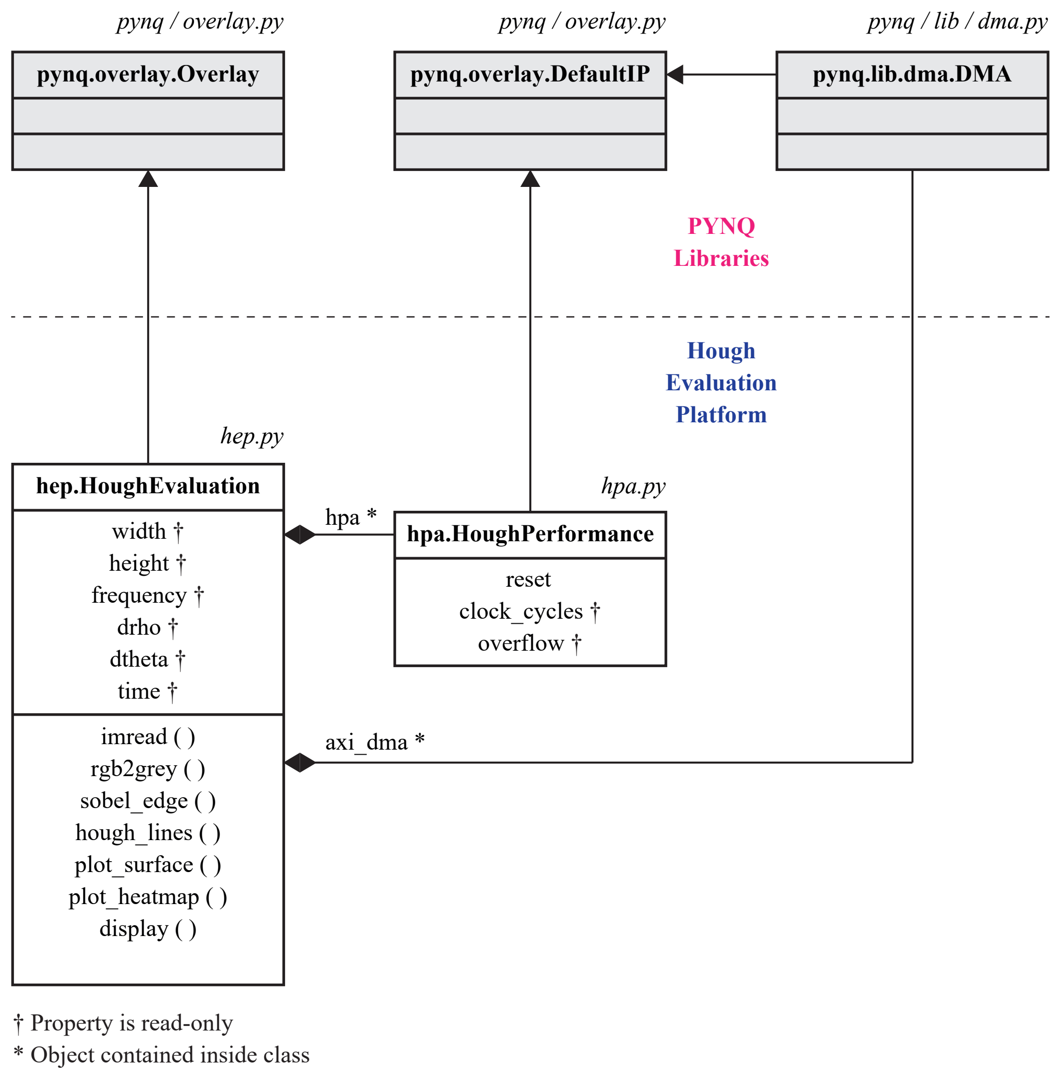


Figure 4.6: A UML diagram representing the properties, methods, and classes of the HEP software drivers and APIs.

As described in the PYNQ documentation, when the *Overlay* class is initialised, it searches known library files for a special property named `bindto`. This property will contain a vendor, library, name, and version (VLNV) string that is used to bind a Python driver to an IP core with a matching VLNV in the IP Integrator design. This process is used to bind the custom *HoughPerformance* class to the HPA IP core. The remainder of this section will describe the custom Python classes for the HEP. Complete source code listings can be found electronically in [106].

The HoughEvaluation Class

The *HoughEvaluation* class is stored in *hep.py* and inherits from the PYNQ *Overlay* class [110], which maintains track of the bitstream's state and the contents of the FPGA. The *Overlay* class is able to expose IP cores to the user and bind software drivers to IP cores as required. The class is also able to program the FPGA with a bitstream.

The *HoughEvaluation* class is configured to store information about the HEP. The user must initialise the class by specifying the acceptable width and height of the LHT architecture, and the system clock frequency. Optionally, the user may also specify the value of δ_ρ and δ_θ, which are named *drho* and *dtheta* in the code, respectively. There are several methods that have been added to the *HoughEvaluation* class to allow users to easily perform common tasks and processes. Methods include colour conversion, Sobel edge detection, Plotly surface and heatmap plots, and an LHT method that executes the user's custom LHT architecture.

Also included in the *HoughEvaluation* class are the *axi_dma* and *hpa* objects that interact with the underlying hardware design. The *axi_dma* is an object of type *DMA*, which controls the AXI DMA IP core in the FPGA logic fabric. The driver is provided in the core PYNQ library in the *dma.py* source file [111]. The HEP uses already existing methods in the *DMA* class to interact with the AXI DMA and control the transfer of data between the Zynq's PS and PL. Notably, the *DMA* class inherits properties and methods from PYNQ's *DefaultIP* class [112]. The *DefaultIP* class is very useful for creating new IP core drivers as it provides AXI4-Lite register read and write methods.

The HoughPerformance Class

The *HoughPerformance* class also inherits the *DefaultIP* class to simplify its design. Recall previously in Section 4.2.2 that the HPA IP core contains three AXI4-Lite registers. These are the reset, clock cycles, and overflow registers. As presented in Figure 4.6, the *HoughPerformance* class contains three corresponding properties. Both the clock cycles and overflow registers are read-only. The clock cycles property can be read to obtain the number of clock cycles required to complete architecture operation. The reset property can be controlled by the user to reset the HPA after operation.

4.3.4 HEP System Integration Workflow

A flow diagram illustrating the HEP system design workflow is shown in Figure 4.7. There are five main stages: planning, HDL Coder design, Vivado integration, PYNQ integration, and system redesign. The workflow begins at the specification stage and ends after application execution and testing.

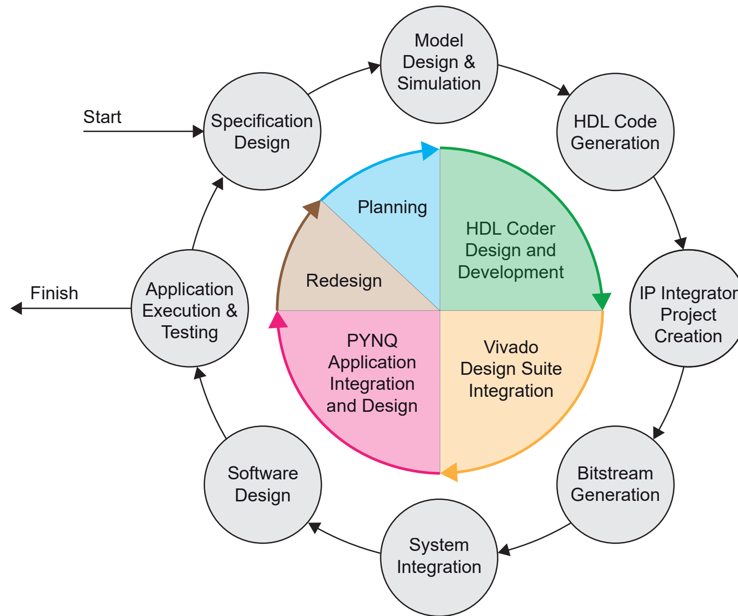


Figure 4.7: A diagram depicting the HEP embedded system design workflow.

The properties of the LHT architecture are defined in the specification stage. The architecture is then modelled and simulated using *HDL Coder*. When the architecture is ready, HDL code is generated and an IP Integrator block design is created. The FPGA bitstream is generated and the PYNQ framework is initialised with the hardware design. The user can add their own software and test their LHT application. If the LHT system fails to meet expectations, the architecture can be redesigned. Both Vivado and *HDL Coder* are essential, as their combined capabilities allow designers to rapidly generate FPGA systems for Zynq MPSoC devices. PYNQ enables the visualisation and analysis capabilities previously described in Section 4.2.3. The remainder of this section presents a typical workflow using the HEP, which begins with a Simulink template model for architecture design and simulation.

The HEP Simulink Template

To simplify system integration, the user is provided with a Simulink template model to create their custom LHT architecture. The template model is preconfigured with appropriate HDL properties that bind to the HEP reference design. The Simulink template model is presented in Figure 4.8 and contains four subsystems.

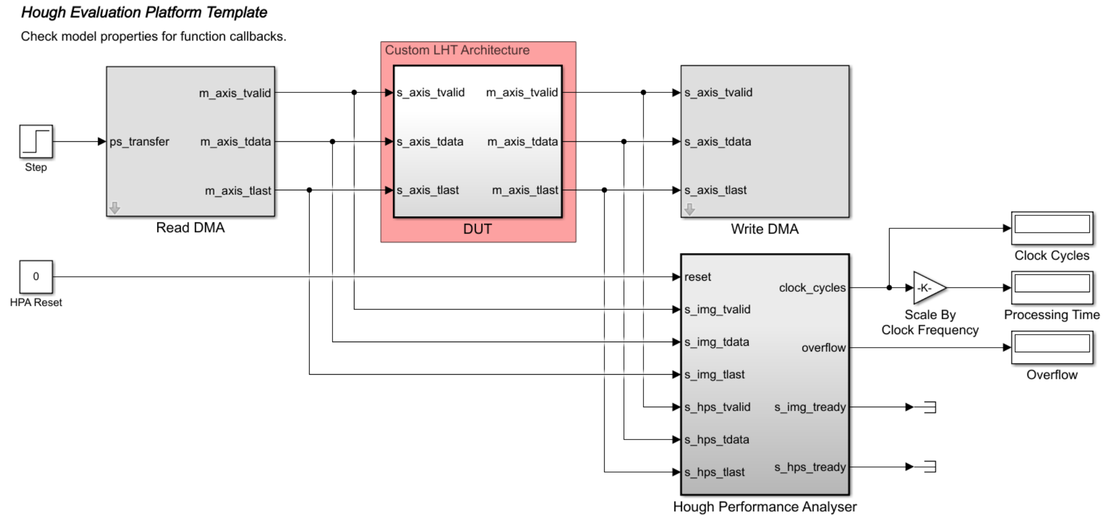


Figure 4.8: The Simulink reference design template to simplify HEP integration.

The DUT subsystem (also known as the Design Under Test) will contain the user's custom LHT architecture design. The Read DMA subsystem simulates an AXI DMA transfer from the PS to the PL. Similarly, the Write DMA subsystem simulates an AXI DMA transfer from the PL to the PS. Lastly, the HPA subsystem measures the architecture's processing time.

The designer configures the DUT subsystem to their custom LHT architecture design. When complete, the model can be simulated to ensure correct operation before system integration. The user can select their own custom images to be transferred into the DUT using the Read DMA subsystem. The image is then transferred by the Read DMA subsystem when the *ps_transfer* input port receives a rising edge. After the image is processed in the DUT, it is transferred to the Write DMA subsystem where it is buffered for analysis. During the simulation, the HPA subsystem measures the number of clock cycles required to complete the architecture operation.

The HDL Workflow Advisor

The plug-in files described in Section 4.3.1 are loaded into *HDL Coder* when it is launched. The plug-in files integrate into *HDL Coder's* Workflow Advisor software tool, allowing the user to follow a series of steps to generate the HEP's bitstream and program the target platform. The HDL Workflow Advisor targets the DUT subsystem presented in Figure 4.8. There are four primary steps that include setting the target platform, preparing the model for HDL code generation, performing HDL code generation, and carrying out embedded system integration. A screenshot of the *HDL Coder* Workflow Advisor is presented in Figure 4.9.

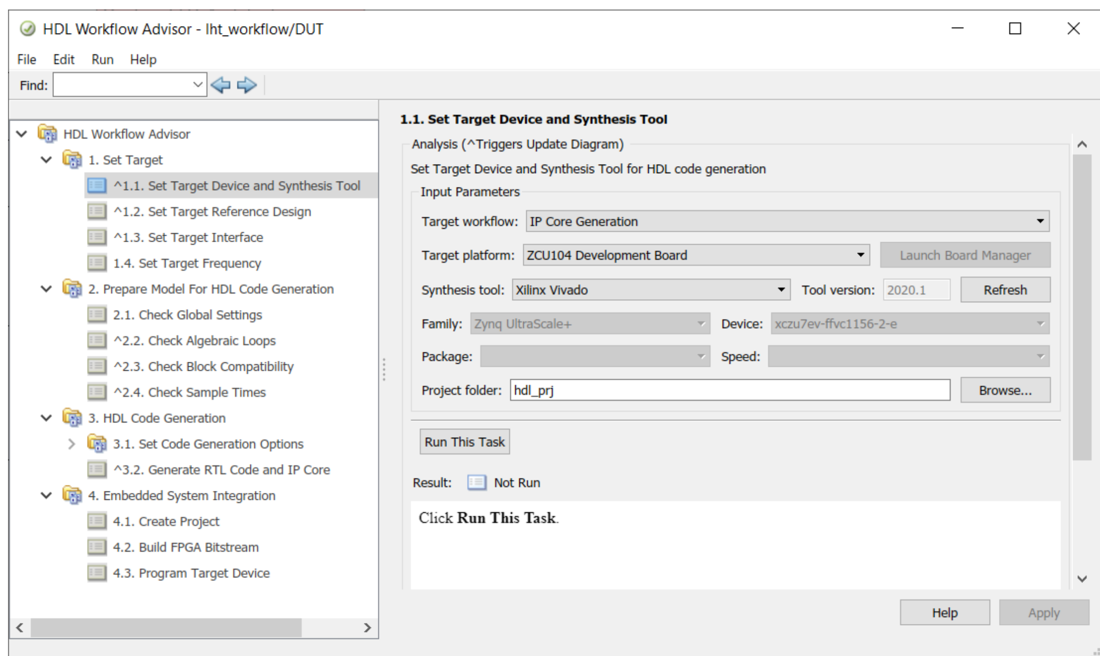


Figure 4.9: A screenshot of the *HDL Coder* Workflow Advisor.

To perform embedded system integration, the user has the option to follow the steps outlined in the HDL Workflow Advisor. This thesis will not explore each step of this process, as these are covered extensively in the documentation provided by MathWorks [113]. Instead, changes and customisations to the workflow that are required to implement the HEP are described. These include setting the board IP address and system clock frequency properties, configuring the DUT's target interface for AXI4-Stream communication, and implementing the custom programming callback step.

Configuring Embedded System Properties

During Step 1.2 of the HDL Workflow Advisor, the user is prompted to enter the IP address of the ZCU104 development platform. The IP address is required later in Step 4.3 to upload the software driver files and bitstream to the Jupyter file system using SSH. A screenshot of Step 1.2 is presented in Figure 4.10. This user prompt is associated with the code given in Listing 4.5.

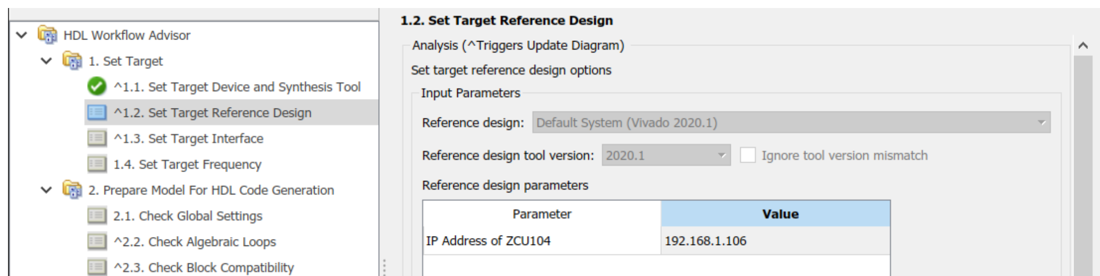


Figure 4.10: A screenshot of the HDL Workflow Advisor during Step 1.2. The board IP address is set for SSH communication.

The system clock frequency, which is used to operate the custom LHT architecture, is configured in Step 1.4 of the HDL Workflow Advisor. As shown in Figure 4.11, the user is prompted to enter a clock frequency between 10 MHz and 250 MHz. This user prompt corresponds to the code presented in Listing 4.4.

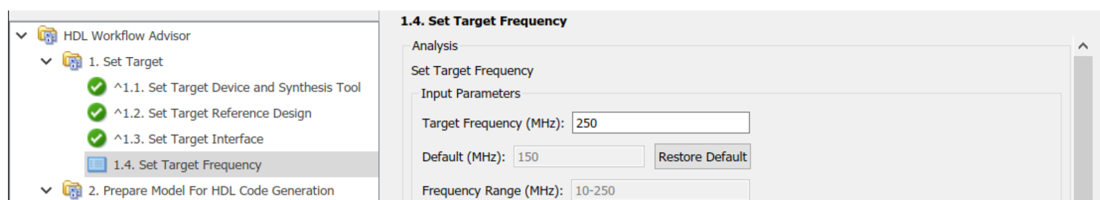


Figure 4.11: A screenshot of the HDL Workflow Advisor during Step 1.4. The system clock frequency is configured.

The code presented in Listing 4.2 creates two new AXI4-Stream interfaces for the HEP reference design. The resulting interfaces are then connected to the AXI4-Stream Broadcasters in the IP Integrator design. This functionality is applied in Step 1.3 of the HDL Workflow Advisor, which is presented in Figure 4.12. The AXI4-Stream signals are applied to their Simulink counterparts in the target platform interface table (highlighted in red).

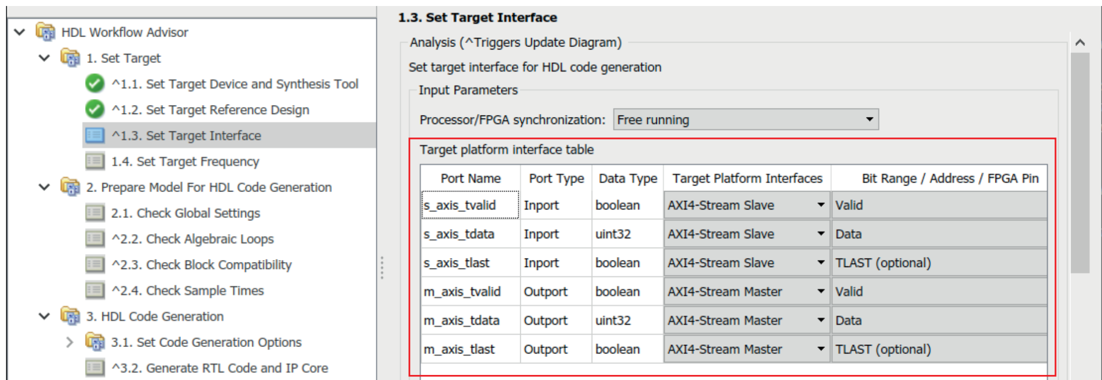


Figure 4.12: A screenshot of the HDL Workflow Advisor during Step 1.3. Setting the target interfaces for HDL code generation.

Programming the Target Platform

The final stage of the HDL Workflow Advisor is programming the target platform. Recall previously in Section 4.3.1 that a custom callback function was created that uploads the software driver files and bitstream into the Jupyter file system. Running Step 4.3 of the HDL Workflow Advisor performs this operation and also opens the Jupyter workspace in a web browser. Figure 4.13 presents the Jupyter workspace after custom programming. Notice the file system contains all of the required software, bitstream, and the default HEP notebook.

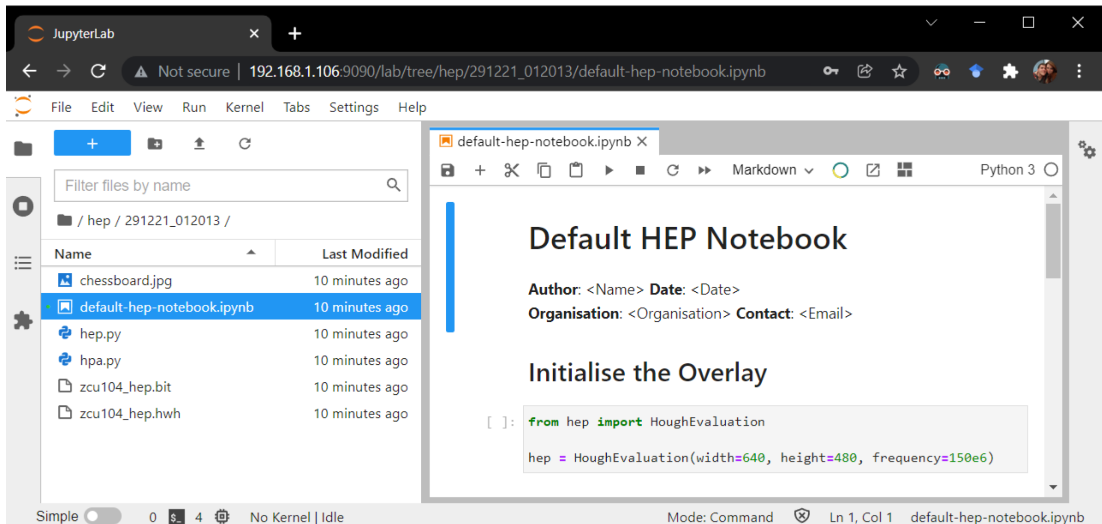


Figure 4.13: A screenshot of the Jupyter environment that is accessed using a web browser. The HEP software drivers, bitstream, and default notebook have been transferred into the file system using SSH.

4.4 Analysis and Evaluation

In this section, the HEP is evaluated for its FPGA resource consumption and timing closure. Furthermore, a parallel LHT architecture was designed to test the HEP's system integration and analysis capabilities. The implementation of the parallel LHT architecture is presented in Appendix B and is based on the work described in [87].

4.4.1 Implementation Results

The Vivado Design Suite was used to synthesise and implement the HEP to evaluate its FPGA resource consumption and timing closure. The HEP was initially implemented separately from the LHT architecture given in Appendix B. The target clock frequency of the HEP was 250 MHz, and the allocated FPGA resources were reported as shown in Table 4.1. The HEP consumes 5 BRAM tiles for the AXI DMA and very few logic fabric resources i.e. FFs, LUTs, and LUT RAM.

Table 4.1: FPGA resource requirements for the HEP on the XCZ7UEV-2E device.

Resource	Available	Used	Percentage (%)
LUTs	230,400	5,792	2.51
LUT RAM	101,760	1,035	1.02
FFs	460,800	8,944	1.94
BRAM	312	5	1.60
DSP48E2	1,728	0	0.00

The LHT architecture given in Appendix B was implemented on the ZCU104 platform using the HEP. The architecture was designed to apply the LHT to an image of 1920×1080 pixels and the discrete step across the HPS axes was set to $\delta_\rho = 1$ and $\delta_\theta = 1°$. The LHT architecture was able to achieve a maximum clock frequency of 200 MHz. The FPGA resource allocation was reported as shown in Table 4.2.

Table 4.2: FPGA resource requirements for the HEP and LHT on the XCZ7UEV-2E device.

Resource	Available	Used	Percentage (%)
LUTs	230,400	22,553	9.79
LUT RAM	101,760	1,003	0.99
FFs	460,800	17,496	3.80
BRAM	312	275	88.14
DSP48E2	1,728	89	5.15

The parallel Hough kernel uses 89 DSP48E2 slices to implement (3.2). The number of memory bits required to store the HPS for an image of 1920 × 1080 pixels is 4,790,640 bits, which is calculated using (3.5). If the HPS was mapped directly to BRAM resources, without considering circuit limitations and throughput, the number of BRAM tiles required would be 130. However, the FPGA resources allocated to the accumulator memory is 270 BRAM tiles using the XCZ7UEV-2E device. Each accumulator subsystem consumes one 36 Kb BRAM and one 18 Kb BRAM to store the votes for one angle in θ. The BRAM resources to store the HPS for the parallel LHT architecture are significantly overallocated.

The parallel LHT architecture was generated for several image resolutions to evaluate its memory consumption compared to the standard LHT. These architectures can be investigated electronically in [106]. Table 4.3 presents the total memory bits to store the HPS for the standard LHT and parallel LHT. The standard LHT memory requirements were computed using (3.5). The parallel LHT memory consumption was determined by the number of BRAM tiles required to implement the architecture on the XCZ7UEV-2E device. Note that $\delta_\rho = 1$ and $\delta_\theta = 1^\circ$ for each implementation.

Table 4.3: The HPS memory consumption of the standard LHT and the parallel LHT.

Resolution (Pixels)	**Standard LHT**	**Parallel LHT**			**Memory Over Allocated(%)**
	Total Bits	**BRAM**		**Total Bits**	
		18 Kb	**36 Kb**		
320 × 240	648,000	180	—	3,317,760	512.00
333 × 333	764,640	180	—	3,317,760	433.90
512 × 512	1,306,800	180	—	3,317,760	253.88
800 × 600	1,800,000	180	—	3,317,760	184.32
1024 × 768	2,534,400	—	180	6,672,384	263.27
1280 × 720	2,910,600	—	180	6,672,384	229.24
1920 × 1080	4,760,640	180	180	9,953,280	209.07

The parallel LHT is memory inefficient for small image resolutions. For example, when applying the LHT to an image of 320 × 240 pixels, the parallel LHT allocates 3,317,760 bits of memory using 180 18 Kb BRAMs. However, the standard LHT requires at least 648,000 bits to store the HPS. The parallel LHT inefficiently over allocates 512% of the necessary memory resources, as shown in the final column of Table 4.3. The other image resolutions also exhibit similarly inefficient allocations of BRAM tiles.

4.4.2 Processing Time Results

The HEP can calculate the processing time of the parallel LHT architecture presented in Appendix B. The processing time of this architecture for one 1920×1080 pixel image is reported by the HEP as 12.35 ms. This processing time corresponds to a frame rate of approximately 80.96 fps. This frame rate is suitable for applying the LHT to the FHD video standard, which uses 1920×1080 pixels per video frame in progressive display mode at 60 fps. The parallel LHT architecture was also hardware validated using the HEP for other image resolutions. The processing time results are presented in Table 4.4 and are rounded to two decimal places.

Table 4.4: Processing time results of parallel LHT architectures that target various image resolutions. The processing time column contains measurement results from the HEP.

Resolution (Pixels)	**Clock Frequency (MHz)**	**Processing Time (ms)**	**Frames Per Second (fps)**
320×240	250.00	0.60	1,666.67
333×333	250.00	0.79	1,265.82
512×512	250.00	1.57	636.94
800×600	250.00	2.64	378.79
1024×768	225.00	4.52	221.24
1280×720	225.00	5.27	189.75
1920×1080	200.00	12.35	80.96

The fastest processing time is achieved by the smallest image resolution, which is 320×240 pixels. The parallel LHT architecture can process this image resolution in 0.60 ms, corresponding to 1,666.77 fps. This frame rate is very suitable for safety critical tasks required by the embedded applications discussed previously in Section 3.4.

4.4.3 Architecture Validation and Testing

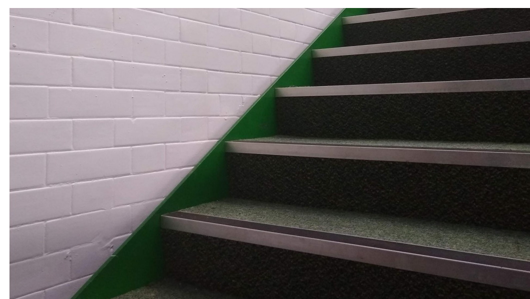
Software validation is performed by comparing the output HPS of the LHT architecture with the results of an LHT software model designed in MATLAB. The source code for this software model is presented in Appendix B and operates by applying the LHT to a candidate test image to produce an output HPS. The resulting HPS from the architecture design and the output HPS from the software model are directly compared. When both models produce the same HPS across one or more candidate test images,

the LHT architecture is ready for HDL code generation and system integration. The author of this thesis recognises that robust tests are required to achieve application specifications and constraints. However, the software validation approach described above is suitable for research and development purposes.

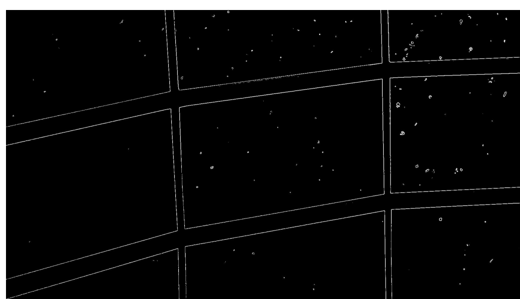
The primary advantage of using the HEP is its ability to validate LHT architecture designs on the physical target device. After bitstream generation, code automation is performed where all necessary system files are transferred into the Jupyter environment on the ZCU104 development board. From here, the user can perform hardware validation using the LHT architecture to process an input test image. The resulting HPS can then be compared with the same software model used previously during software validation or with a new software model in the Jupyter environment.

The parallel LHT architecture design was successfully validated on hardware using two different test images. These images and their edge-detected representations are given in Figure 4.14.

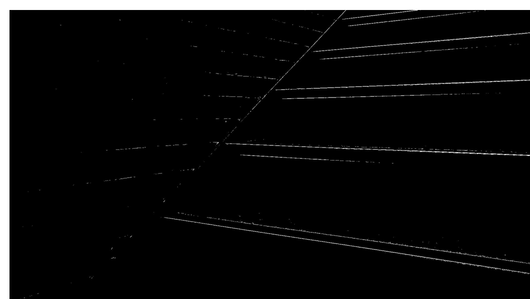
(a) Colour image of a window.



(b) Colour image of a set of stairs.



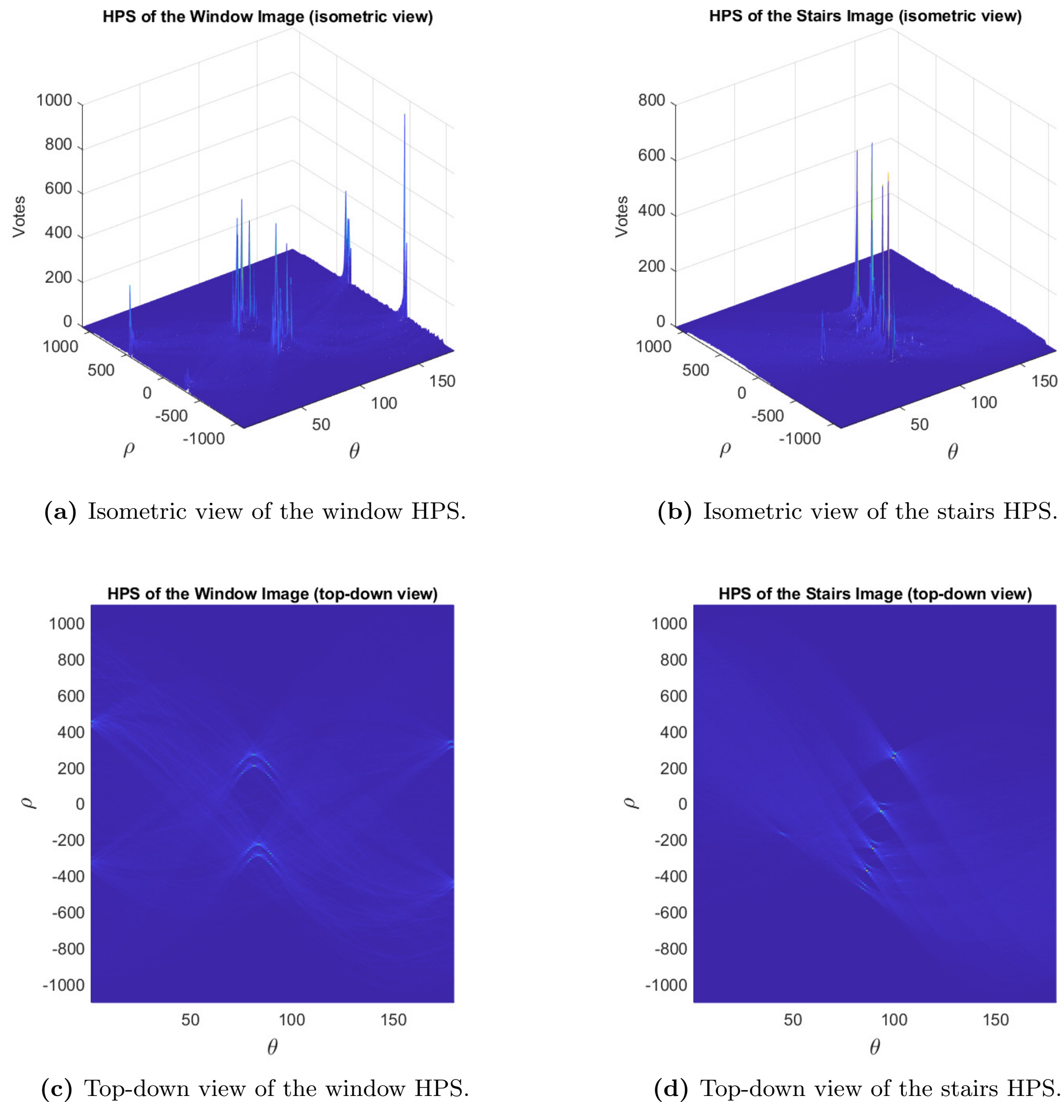
(c) Edge image of the window.



(d) Edge image of the set of stairs.

Figure 4.14: Two test images (a) and (b). Their corresponding edge images (c) and (d) using Sobel edge detection with a threshold of 80 and 70, respectively.

The parallel LHT architecture and software model of the LHT both return the HPS of an input image. For each test image in Figure 4.14, the HPS returned by the simulation and software model of the LHT were compared using element-wise comparison techniques. This comparison involves simultaneously iterating through the elements of both HPS arrays and applying the equality operator to determine if both arrays are equal. Note that the equality operator is '==' for most programming languages. The resulting HPS for each image is given in Figure 4.15.



(a) Isometric view of the window HPS.

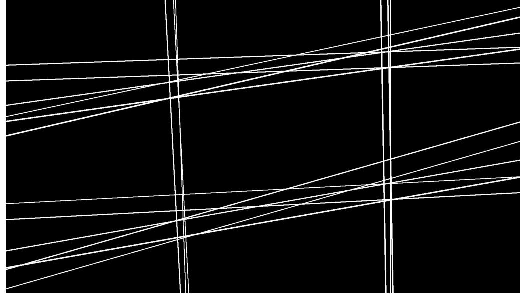
(b) Isometric view of the stairs HPS.

(c) Top-down view of the window HPS.

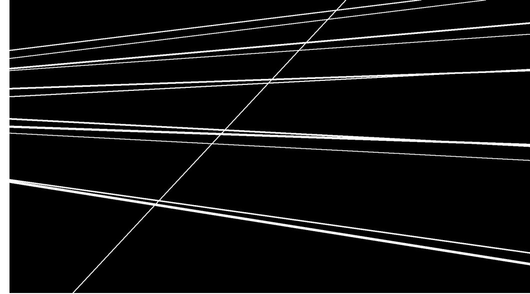
(d) Top-down view of the stairs HPS.

Figure 4.15: HPS results for the hardware validation of the parallel LHT architecture on the XCZU7EV-2E device. The isometric view of the HPS for the window image (a), and the stairs image (b). The top-down view of the HPS for the window image (c), and the stairs image (d).

Finally, the parameters of lines are obtained from the HPS for each test image and are reconstructed using (3.6) and (3.7). Figure 4.16 presents the line reconstruction results for each input test image.

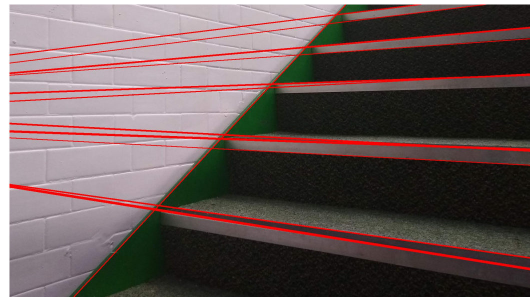


(a) Reconstructed line image of the input window edge image.



(b) Reconstructed line image of the input stairs edge image.

(c) Overlay of the reconstructed image and the original colour image of the window.



(d) Overlay of the reconstructed image and the original colour image of the stairs.

Figure 4.16: Line reconstruction results for the test images input into the parallel LHT architecture. The reconstructed line images of the window (a) and stairs (b). The reconstructed lines overlaid on top of the original colour images of the window (c) and stairs (d).

Figure 4.16a and Figure 4.16b contain the reconstructed line images for the window and stairs, respectively. The reconstructed line images are overlaid on top of the original colour images for inspection in Figure 4.16c and Figure 4.16d. Notice that the lines correspond with collinear features in the original test images.

The HEP was also used to validate the LHT architecture using the Jupyter environment, as shown in Figure 4.17 for the window image and Figure 4.18 for the stairs image. The resulting HPS for each test image was the same as that produced by the LHT simulation and MATLAB software model. This comparison was performed using the same procedure used to compare the HPS arrays produced by the simulation and software model of the LHT.

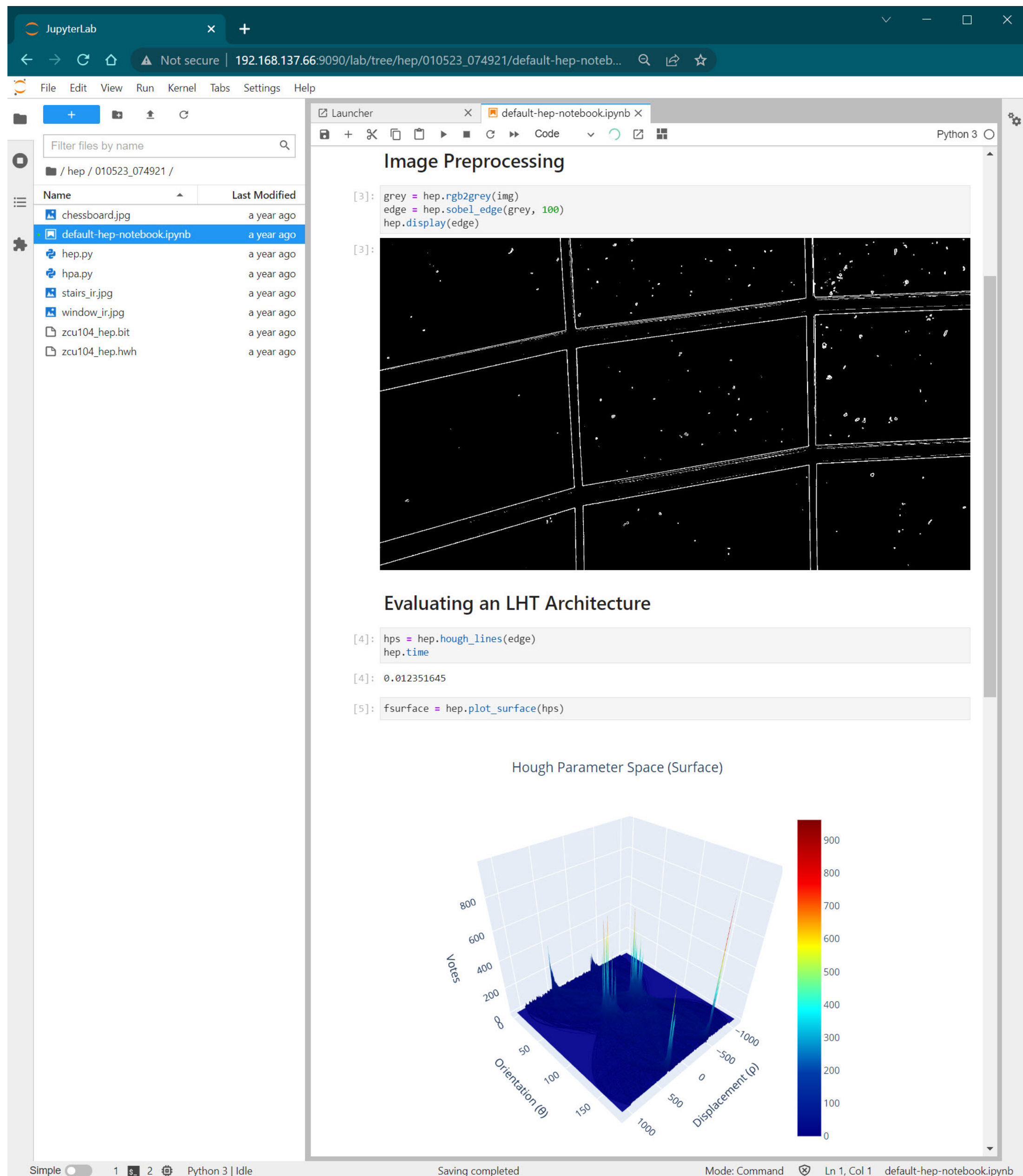


Figure 4.17: This screenshot presents the HEP Jupyter environment, where the LHT architecture is undergoing hardware validation using the window image.

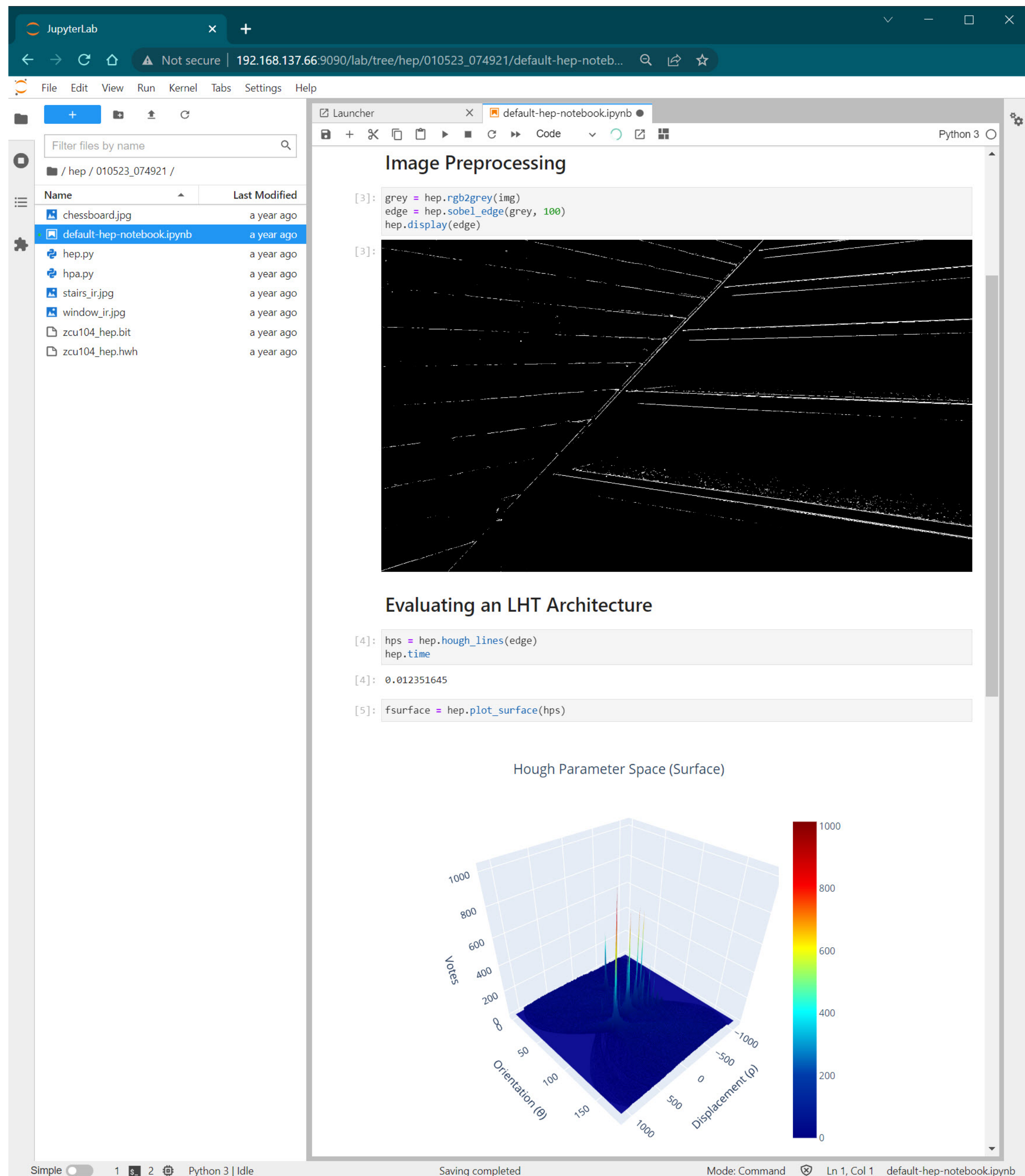


Figure 4.18: This screenshot presents the HEP Jupyter environment, where the LHT architecture is undergoing hardware validation using the stairs image.

4.4.4 Comparison with FPGA-in-the-Loop Simulation

Upon reviewing previously published literature, it was found that the HEP is the first open-source tool of its kind. The HEP combines PYNQ and MathWorks HDL Coder, performs on-target validation of FPGA architectures, and accurately computes the processing time of implemented designs. However, there is a verification technique for FPGA architectures known as an FPGA-in-the-loop (FIL) simulation that is worth comparing against to establish the significance of the HEP as a novel contribution.

An FIL simulation is an established technique for verifying the functionality of an FPGA architecture design. The basic principles of an FIL simulation involve implementing an architecture design on a target FPGA device that may interface with other physical instruments or components. The FPGA architecture design also integrates with a simulation model of a more extensive system operating on a host computer. Researchers and developers generally use the simulation model to inject testing stimulus into the FPGA architecture over an interface such as a Joint Test Access Group (JTAG) connection or Ethernet. MathWorks has a commercial solution for performing FIL simulations, which has extensive documentation in [114].

The HEP offers researchers and developers several advantages compared to an FIL simulation environment. These advantages are listed as follows.

1. *External Interfaces* — Researchers and developers can generate and store test data using the HEP's Jupyter Lab environment, which executes on the Zynq MPSoC's PS. Compared to FIL simulations, this capability of the HEP is advantageous as it is not necessary to use an external interface to transfer data between a host computer and the target device. This feature of the HEP allows researchers and developers to readily deploy their FPGA architecture designs in a testing scenario without connecting externally to a host computer.

2. *Communication Overhead* — During FIL simulations, the target device and host computer often exchange test data using an external interface. This communication interface introduces latency and overhead, which may not reflect the true behaviour of the FPGA architecture design when it operates outside of FIL sim-

ulations. In contrast, the HEP does not have this issue, as it can transfer test data to and from the Jupyter Lab environment at the intended execution speed of the FPGA architecture.

3. *Realistic Execution* — FIL simulations provide a near hardware-accurate testing environment. However, due to communication overhead, they cannot capture the realistic behaviour of the FPGA architecture design in a deployed system. In contrast, the HEP does not have communication overhead. This capability of the HEP is essential to researchers and developers as the processing time of the FPGA architecture can be computed accurately during execution.

The HEP offers significant advantages compared to FIL simulations and improves FPGA architecture verification capabilities. However, FIL simulations are readily supported across several FPGA devices, while the HEP is currently supported on one development board (the ZCU104). Adding HEP support for other development boards requires engineering and development time. Furthermore, the HEP is only available for the Zynq MPSoC and Zynq-7000, as these devices have PYNQ support.

4.4.5 Discussion of Results

The primary motivation of this thesis is to reduce the memory requirements of the LHT for FPGA devices. To begin exploring the memory consumption of the HPS, a suitable development environment, design and testing framework, and FPGA integration workflow are required. As demonstrated in this chapter, the HEP provides all of these requirements and is very effective in rapidly producing LHT architecture designs. Furthermore, this work is the first FPGA development and evaluation environment that combines PYNQ and MathWorks *HDL Coder*. Note that the HEP could also be used to develop and evaluate FPGA architectures for other image and signal processing applications beyond that of LHT circuit designs. However, this capability of the HEP is not the focus of the work presented in this thesis and would require further engineering and development time depending on the target application and algorithm.

A significant outcome of this chapter is the design and testing framework that has

emerged as a result of creating the HEP. For instance, LHT architectures are captured using MathWorks *HDL Coder* blocks and are simulated to verify architecture operation. Software simulations of an LHT architecture are useful as they can help identify design errors before synthesis and implementation, which are particularly time-consuming. As described in Section 4.4.3, architectures can be verified on the physical target device by comparing hardware results with corresponding software models. This verification technique improves the testing of hardware architectures beyond software simulations and provides an effective technique for architecture validation.

The system integration workflow that is described in Section 4.3.4 is another important outcome of this work. When a new LHT architecture is in development, each design variation undergoes a similar development framework, which is presented in Figure 4.7. Maintaining a robust development workflow reduces the risk associated with design errors, which may result in incorrect research claims and results.

Finally, the implementation of the LHT on the XCZ7UEV-2E device has been enabled by the HEP and its underlying development framework. There are many significant findings in this work, which include FPGA resource consumption reports and processing time analysis of the parallel LHT architecture described in Appendix B. The primary issue identified with this LHT architecture design is the large memory requirement of the accumulator array. The efficiency of the HPS needs to be improved to decrease on-chip memory allocation. The remainder of this thesis investigates on-chip memory consumption and FPGA resource allocation of the LHT.

4.5 Conclusion

A system overview and design specification of the HEP was described at the beginning of this chapter. Subsequently, an architecture overview of the HIU and HPA was provided. Several components of the HIU, including the AXI DMA, AXI4-Stream Broadcaster, and the HPA were explored. The software components of the HEP were then described. These components included the visualisation and analysis capablities of Jupyter Labs using the Plotly Python library, and the processing time analysis of the user's custom LHT architecture design.

After describing several features of the HEP, the MathWorks *HDL Coder* reference design was presented. Several aspects of its design were investigated, including its plug-in files and automated programming workflow. The IP Integrator block design was then explored, alongside a UML diagram representing the properties, methods, and classes of the HEP's software drivers and APIs. Finally, the HEP system integration workflow, template, and programming steps were described. Analysis and evaluation of the HEP and a parallel LHT architecture were carried-out. Results concluded that the HEP was able to achieve a target clock frequency of 250 MHz with very low resource consumption. The LHT architecture achieved a maximum clock frequency of 200 MHz. This architecture consumed 270 BRAM tiles and 89 DSP48E2 slices on the XCZ7UEV-2E device when targeting an image resolution of 1920×1080 pixels.

A notable output of this work is the open-sourcing of the HEP, which is now a software tool that is freely available online [106] to download. At the time of writing, the HEP is regularly downloaded by approximately five unique users per week. This level of community engagement is high for a niche evaluation platform targeting the LHT. The HEP will allow other researchers in the community to design and evaluate their own LHT architectures. There are also three primary findings and outcomes from the work undertaken in this chapter. These are listed as follows:

1. The HEP and its design was described and the LHT architecture development framework was explored. For the remainder of this thesis, the HEP will be used to rapidly design and evaluate the LHT architectures presented in Chapter 5 and Chapter 6.

2. To ensure the LHT architectures are designed correctly in subsequent chapters, their testing and hardware validation can be performed using the HEP. This design and testing methodology will ensure that all architecture designs can be compared fairly and with research integrity.

3. The FPGA resource consumption of the parallel LHT architecture in Appendix B was significantly higher than that required by the standard LHT. In particular, BRAM resources were inefficiently allocated to the accumulator array, leaving

several memory locations unused. In Chapter 5, a bit-packing scheme is employed to reduce BRAM memory consumption without compromising on the LHT's configuration and design specification.

Chapter 5

A Symmetric Hough Kernel and Bit-Packed Accumulator

5.1 Introduction

Parallel LHT architectures require many specialised FPGA resources to implement the Hough kernel and accumulator memory. A significant issue with LHT architecture designs that process large image resolutions is their inefficient allocation of memory resources to store the HPS. This inefficiency often leads to the overconsumption of on-chip FPGA resources, such as BRAMs and LUT memories. Consequently, inadequate accumulator designs require FPGA devices with significant memory resources.

This chapter presents a novel technique to reduce the resource consumption of the accumulator memory in LHT architecture designs. This technique is named the Symmetric LHT, as it exploits symmetry in the spatial image domain to reduce computation and decrease on-chip memory allocation. Furthermore, resource-sharing techniques are employed to decrease the arithmetic resources allocated to the Hough kernel. It is important to note that the Symmetric LHT algorithm and architecture can apply the LHT to an image without affecting the accuracy of line extraction, i.e. it produces the same results as the standard LHT in [3]. Finally, this chapter describes the Symmetric LHT architecture design and presents its FPGA resource consumption and processing time results as reported by the Vivado Design Suite and the HEP, respectively.

5.2 The Symmetric LHT

This section describes the Symmetric LHT algorithm and evaluates its memory and arithmetic resource consumption in AMD FPGA devices. The Symmetric LHT algorithm contains five stages. Figure 5.1 presents an overview of each stage using a functional block diagram.

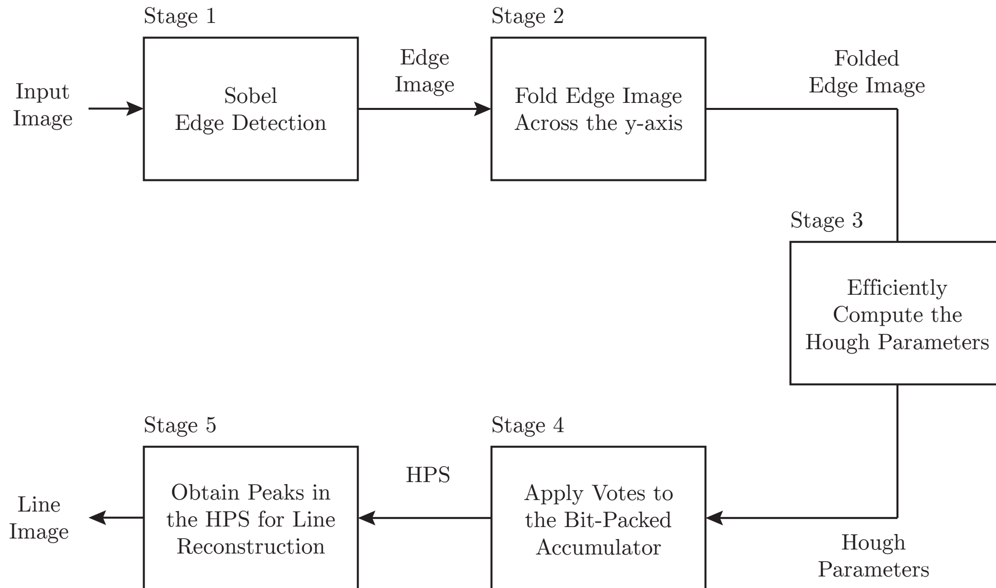


Figure 5.1: Functional block diagram of the Symmetric LHT, which presents five main stages of the algorithm.

Initially, edge detection is applied to an input image at Stage 1 of the algorithm. Stage 2 reduces the computational requirements of the LHT by folding the input edge image across the y-axis. Stage 3 efficiently calculates the Hough parameters for two edge pixels that are symmetrical across the y-axis using only one set of image coordinates. In Stage 4, the Hough parameters vote and form peaks in the bit-packed accumulator array. The final stage performs peak extraction in the HPS to obtain Hough parameters corresponding to lines within the input image. The remainder of this section provides further detail about each stage of the Symmetric LHT. The next section will briefly describe FPGA resource sharing, as this is an important technique used to design the Symmetric LHT architecture.

5.2.1 Resource Sharing

FPGA architecture designs operate at a specified clock frequency, commonly denoted as f_{clk}. Architecture designs can operate at sample frequencies equal to f_{clk} or at an integer sub-multiple of f_{clk} using clock-enable logic. Architectures that have more than one sample rate are known as multirate systems. The Symmetric LHT uses a common FPGA architecture design technique known as resource sharing, which exploits multirate systems to optimise resource consumption [115]. Resource sharing is performed by time multiplexing signals into an FPGA resource that performs the same arithmetic operation on each input signal. To explain this technique, consider the FPGA architecture design in Figure 5.2, which multiplies two inputs by a constant value.

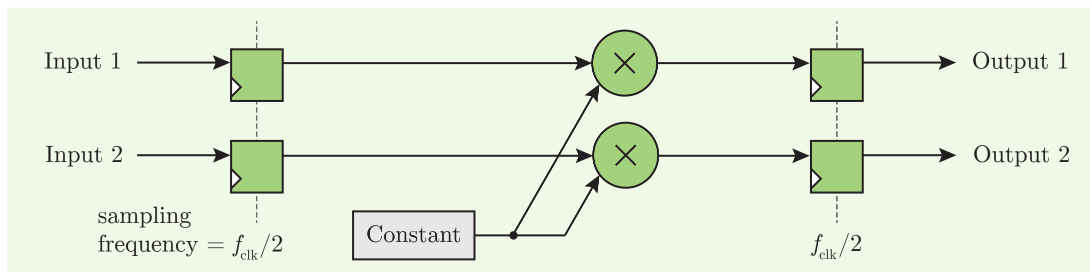


Figure 5.2: An FPGA architecture that multiplies two input signals by a constant value.

The input signals use a sampling frequency that is half of the architecture's clock frequency, i.e. $f_{\text{clk}}/2$. In this architecture, there are two separate multipliers that each perform a multiplication operation for each input signal. It is possible to use one multiplier in this design through resource-sharing techniques. Figure 5.3 presents a resource-shared version of the above architecture.

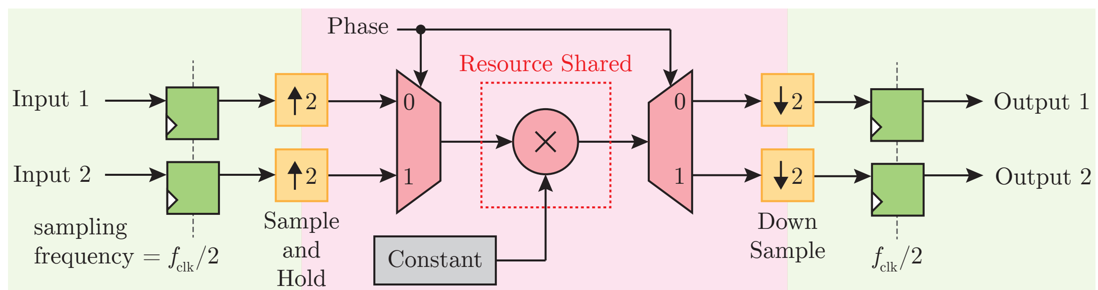


Figure 5.3: An FPGA architecture that exploits resource sharing to multiply two input signals by a constant value using one multiplier.

The sampling frequency of each input signal is half of the architecture's operational clock frequency. The multiplier in the design can operate at the higher sample frequency, which is equal to the architecture's clock frequency, f_{clk}. The input signals are time multiplexed into the multiplier using the phase signal to swap between each input. It is only possible to use resource-sharing techniques when the sampling frequency of the input signals are lower than the architecture's clock frequency by an integer factor. The Symmetric LHT architecture design uses resource sharing to approximately half the number of FPGA resources required to compute the Hough parameters. This reduction in complexity is described in the following section.

FPGA resource sharing will be highlighted in architecture designs of the Symmetric LHT for the remainder of this chapter. The colour convention presented in Figure 5.3 will be used to indicate different sample frequencies in a multirate architecture design. The colours and their representative sample frequencies are listed below.

- Red/Pink — These colours represent the maximum sample frequency of the architecture, which is f_{clk}.
- Green — This colour is used to indicate a sample frequency of $f_{\text{clk}}/2$.
- Orange — This colour indicates a rate change in the architecture design.

The Symmetric LHT architecture design does not require a sample frequency lower than $f_{\text{clk}}/2$. Therefore, only three colours are required.

5.2.2 Spatial Domain Symmetry

The standard LHT equation in (3.2) uses image coordinates to calculate a line's magnitude of displacement ρ from the image origin. It is possible to reduce the total computation required to calculate ρ by exploiting the symmetry of coordinates in the spatial domain. Consider the illustration in Figure 5.4, which contains two 10×10 edge images. Each edge image is annotated with axis labels to clearly show their coordinate systems. The edge image on the left contains the coordinate system used by the parallel LHT architecture described in [87], while the right edge image contains the coordinate system used by the Symmetric LHT.

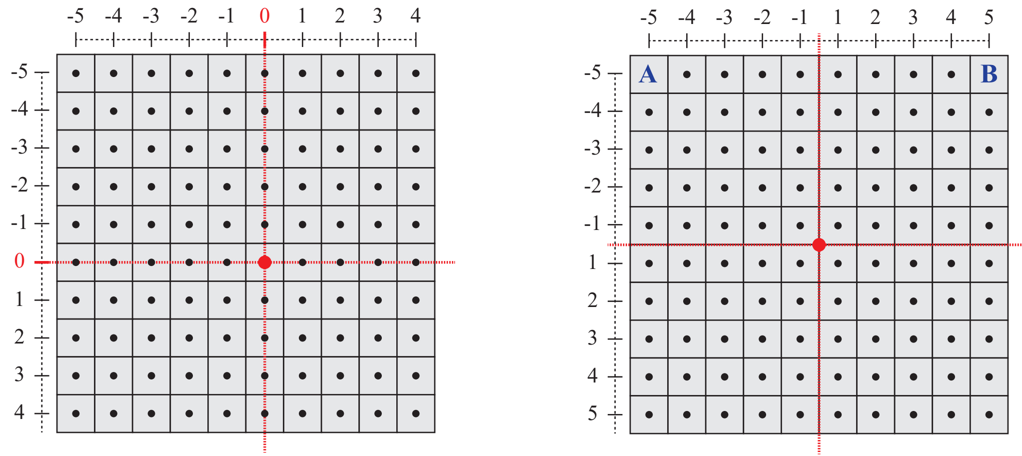


Figure 5.4: Coordinate system used by the parallel LHT architecture (left). Coordinate system used by the Symmetric LHT architecture (right).

The coordinate system on the left of Figure 5.4 is used by most implementations of the LHT. The coordinate system on the right of Figure 5.4 will be used by the Symmetric LHT to reduce computation. This coordinate system contains symmetry over the y-axis when positive and negative signs are ignored. Notice, that there are two points in this coordinate system that are labelled A and B. The coordinates for each point are $(-5, -5)$ and $(5, -5)$, respectively. The Hough parameters can be calculated for each point by using their coordinates in (3.2). However, it is possible to calculate the Hough parameters for each point using only one set of coordinates as

$$\begin{aligned}\rho(\theta) &= x_i \cos(\theta) + y_i \sin(\theta) \\ &= -x_i \cos(180^\circ - \theta) + y_i \sin(180^\circ - \theta).\end{aligned} \tag{5.1}$$

The relationship presented in (5.1) demonstrates that the Hough parameters for points A and B can be calculated using one set of coordinates. This relationship is true for all points of the input edge image that are symmetrical across the y-axis. The width of the input image must be even and the coordinate system presented in the right of Figure 5.4 must be adopted. The FPGA architecture of a symmetric Hough kernel that exploits (5.1) will be able to calculate the Hough parameters for two points of the input image at the same time. The accumulator architecture can leverage this design to bit-pack votes and reduce memory consumption.

An example that demonstrates the relationship in (5.1) will now be presented. Consider points A and B that were shown previously in Figure 5.4. For this example, the Hough parameters for each point will be calculated using (3.2). The Hough parameters for point A and B will be denoted as $\rho_A(\theta)$ and $\rho_B(\theta)$, respectively. The discretisation step of θ will be 22.5°. The Hough parameters for points A and B are presented in Table 5.1 and are rounded to two decimal places.

Table 5.1: LHT results for two points A and B, when $\delta_\theta = 22.5°$.

θ_i	0°	22.5°	45°	67.5°	90°	112.5°	135°	157.5°
$\rho_A(\theta_i)$	-5.00	-6.53	-7.07	-6.53	-5.00	-2.71	0.00	2.71
$\rho_B(\theta_i)$	5.00	2.71	0.00	-2.71	-5.00	-6.53	-7.07	-6.53

The results in Table 5.1 show that $\rho_A(0°) = -\rho_B(0°)$, $\rho_A(22.5°) = \rho_B(157.5°)$, $\rho_A(45°) = \rho_B(135°)$, $\rho_A(67.5°) = \rho_B(112.5°)$, and $\rho_A(90°) = \rho_B(90°)$. These results indicate that the relationship given in (5.1) can be used to reduce the total computation of the Hough kernel architecture in half. The Symmetric LHT algorithm described in this thesis only functions when N_θ is even. It is also worth noting that spatial domain symmetry of the x-axis was not considered during the development of the Symmetric LHT algorithm. It was found that half of the input image would need to be buffered using on-chip memory before x-axis symmetry could be exploited. This operation would consume a considerable amount of memory resources.

The Symmetric LHT also uses the resource-saving techniques presented in [87] to further reduce the number of multiplication operations. This thesis has already presented these techniques, including the efficient Hough kernel illustrated in Figure 3.18 and the Look Ahead Kernel shown in Figure 3.19. The Symmetric LHT uses significantly fewer multiplications than are required by the standard LHT defined in [3]. For example, when θ operates over the range $[0°, 179°]$ and $\delta_\theta = 1°$, the standard LHT requires 360 multiplications. Exploiting spatial domain symmetry reduces the number of multiplications to 180. The efficient Hough kernel in [14] reduces multiplication requirements to 90. Lastly, employing the Look Ahead Kernel in [20] reduces the number of required multiplications to 46. The symmetric Hough kernel architecture design is detailed further in Section 5.3.4.

5.2.3 Parallel Pixel Processing

A common technique to reduce the system clock frequency of an FPGA system is to process two or more samples every clock period. These types of FPGA designs are commonly referred to as Super Sample Rate (SSR) architectures and are often used in high sample rate radio applications, or computer vision systems that require large image resolutions. The Symmetric LHT is able to leverage SSR architecture design to reduce DSP48E2 slice and BRAM tile consumption. Consider the parallel pixel streaming diagram presented in Figure 5.5.

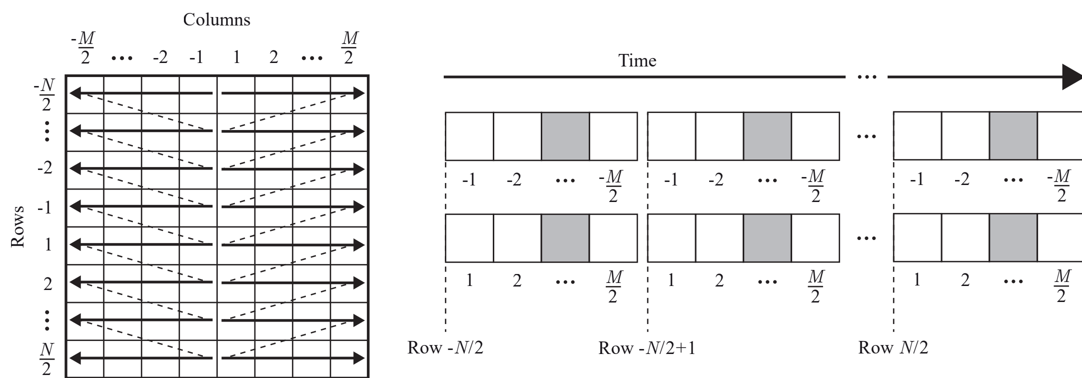


Figure 5.5: Scanning an $M \times N$ image (left), parallel pixel stream of the image (right). Each row of the image is streamed from the image centre to the image borders.

It was previously established in Section 5.2.2 that two symmetric edge pixels across the y-axis can be processed at the same time. The diagram illustrated in Figure 5.5 demonstrates how a candidate image should be streamed through the Symmetric LHT architecture so that symmetry across the y-axis can be exploited.

The diagram shows that the each row of the image is streamed from the image centre to the image borders. This streaming configuration can be achieved by storing the first half of an image row using a First In Last Out (FILO) buffer. As soon as half of the image row is buffered, the last pixel that was pushed into the FILO is popped and can stream alongside the second half of the image row. Note that popping a FILO refers to removing the most recently added element from the FILO. Section 5.3.1 describes the image conditioning and architecture design to create a symmetric pixel processing stream using a FILO, and two FIFO buffers.

5.2.4 The Bit-Packed Accumulator

A significant issue with the parallel LHT architecture is the inefficient allocation of BRAM tiles to implement the accumulator array. The Symmetric LHT can reduce BRAM requirements by bit-packing the HPS. Using the relationship given previously in (5.1), it is possible to store the votes for the Hough parameters $\rho(\theta)$ and $\rho(180° - \theta)$ in the same memory location. The voting space for each of these Hough parameters are concatenated and stored in memory. Table 5.1 presents a platform agnostic memory allocation table to demonstrate the bit-packed memory configuration.

Table 5.2: Platform agnostic memory allocation table for the bit-packed accumulator.

θ **Index**	**Memory Address**	**Stored Data (Bits)**	
		$2b-1$:b	$b-1$:0
θ_0	0	No Data	$A(\rho_0, \theta_0)$
	1	No Data	$A(\rho_1, \theta_0)$
	$\vdots$	$\vdots$	$\vdots$
	$N_\rho - 1$	No Data	$A(\rho_{N_\rho-1}, \theta_0)$
$\theta_1, \theta_{N_\theta-1}$	N_ρ	$A(\rho_0, \theta_{N_\theta-1})$	$A(\rho_0, \theta_1)$
	$N_\rho + 1$	$A(\rho_1, \theta_{N_\theta-1})$	$A(\rho_1, \theta_1)$
	$\vdots$	$\vdots$	$\vdots$
	$2N_\rho - 1$	$A(\rho_{N_\rho-1}, \theta_{N_\theta-1})$	$A(\rho_{N_\rho-1}, \theta_1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\theta_{N_\theta/2-1}, \theta_{N_\theta/2+1}$	$(N_\theta/2-1)N_\rho$	$A(\rho_0, \theta_{N_\theta/2+1})$	$A(\rho_0, \theta_{N_\theta/2-1})$
	$(N_\theta/2-1)N_\rho + 1$	$A(\rho_1, \theta_{N_\theta/2+1})$	$A(\rho_1, \theta_{N_\theta/2-1})$
	$\vdots$	$\vdots$	$\vdots$
	$(N_\theta/2)N_\rho - 1$	$A(\rho_{N_\rho-1}, \theta_{N_\theta/2+1})$	$A(\rho_{N_\rho-1}, \theta_{N_\theta/2-1})$
$\theta_{N_\theta/2}$	$(N_\theta/2)N_\rho$	No Data	$A(\rho_0, \theta_{N_\theta/2})$
	$(N_\theta/2)N_\rho + 1$	No Data	$A(\rho_1, \theta_{N_\theta/2})$
	$\vdots$	$\vdots$	$\vdots$
	$(N_\theta/2+1)N_\rho - 1$	No Data	$A(\rho_{N_\rho-1}, \theta_{N_\theta/2})$

Column one presents the values of θ that are used in a given memory location. Column two presents the memory index, which is used as an address to a memory location. The stored data is presented in columns three and four. The stored data is represented using bit indices, where b is the total number of bits required by each location in the HPS to store votes. The data stored in a memory index are the votes for a location in the HPS, denoted by a location in $A(\rho, \theta)$.

As shown in Table 5.2, the Hough parameters are paired together using $\rho(\theta)$ and $\rho(180° - \theta)$ as a storage pattern. The Hough parameters given by $\rho(\theta_0)$ and $\rho(\theta_{N_\theta/2})$ are stored in their own memories. This design decision can be explained by considering points A and B from Figure 5.4. The memory that stores votes for $\rho(\theta_0)$ will need to be accessed twice to apply votes because, $\rho_A(\theta_0) = -\rho_B(\theta_0)$. The votes for $\rho(\theta_{N_\theta/2})$ are stored in their own memory since, $\rho_A(\theta_{N_\theta/2}) = \rho_B(\theta_{N_\theta/2})$.

So far the bit-packed accumulator has been described without considering its implementation using BRAM tiles. When mapping the bit-packed accumulator to BRAMs, it is essential that three main design decisions are considered before implementation. These are listed as follows.

1. *The partitioning of the bit-packed accumulator* — The bit-packed accumulator will be partitioned over many BRAM tiles. The array should be separated so that all votes can be applied for two symmetric edge pixels. The optimal way to partition the array is across the θ-axis, such that θ_1 and $\theta_{N_\theta-1}$ are stored together in memory, θ_2 and $\theta_{N_\theta-2}$ are stored together in memory, and this pattern is repeated until all θ are appropriately allocated memory.

2. *The BRAM configuration* — A BRAM tile should be configured to use a 36 Kb primitive (a full BRAM tile), or an 18 Kb primitive (a half BRAM tile). The BRAM primitive should be selected based on address requirements to reduce memory allocation. The BRAM should be configured to operate in SDP mode.

3. *The selection of BRAM shapes* — It may be effective to combine different BRAM shapes to reduce memory allocation. For example, a memory array may require 2048 addresses $\times$ 24 bits. It is possible to implement this memory using two 36 Kb BRAM tiles, which are configured to store 1024 addresses $\times$ 36 bits. However, it is more effective to use one 36 Kb BRAM that is configured to store 2048 addresses $\times$ 18 bits, and one 18 Kb BRAM that is configured to store 2048 addresses $\times$ 9 bits. This configuration saves one 18 Kb BRAM.

The following sections will present an example of mapping an accumulator array to BRAM memory. The image resolution used in the example is 1920×1080 pixels. The HPS will be configured using the following parameters: $\delta_\theta = 1°$, $\delta_\rho = 1$, and θ will be over the range $[0, 179]$. These parameters will configure the accumulator for $N_\rho = 2204$, $N_\theta = 180$, and $b = 12$ bits.

Accumulator Mapping to BRAM Tiles

The bit-packed accumulator is mapped to BRAM by first partitioning it across the θ-axis. The votes for θ_0 and θ_{90} will each be stored in their own memories. Both memories must be able to store 2204 addresses × 12 bits, which can be achieved using one 36 Kb BRAM tile and one 18 Kb BRAM. The 36 Kb BRAM tile is configured to store 2048 addresses × 18 bits, and the 18 Kb BRAM is configured to store 1024 addresses × 18 bits. The remaining votes for $\theta_1, \theta_2 ... \theta_{89}$ and $\theta_{179}, \theta_{178} ... \theta_{91}$ are bit-packed into their own memories. A BRAM allocation table is presented in Table 5.3 for each bit-packed memory. The variable i for this particular example is in the range $[1, 89]$.

Table 5.3: BRAM allocation table for a memory instance of the bit-packed accumulator when $N_\rho = 2204$, $N_\theta = 180$, and $b = 12$ bits.

BRAM Configuration	BRAM Address	Stored Data (Bits)		
		[35:24]	[23:12]	[11:0]
36 Kb BRAM: 1024 addresses × 36 bits	0	Zero Pad	$A(\rho_0, \theta_{N_\theta - i})$	$A(\rho_0, \theta_i)$
	1	Zero Pad	$A(\rho_1, \theta_{N_\theta - i})$	$A(\rho_1, \theta_i)$
	⋮	⋮	⋮	⋮
	1023	Zero Pad	$A(\rho_{1023}, \theta_{N_\theta - i})$	$A(\rho_{1023}, \theta_i)$
36 Kb BRAM: 1024 addresses × 36 bits	0	Zero Pad	$A(\rho_{1024}, \theta_{N_\theta - i})$	$A(\rho_{1024}, \theta_i)$
	1	Zero Pad	$A(\rho_{1025}, \theta_{N_\theta - i})$	$A(\rho_{1025}, \theta_i)$
	⋮	⋮	⋮	⋮
	1023	Zero Pad	$A(\rho_{2047}, \theta_{N_\theta - i})$	$A(\rho_{2047}, \theta_i)$
18 Kb BRAM: 512 addresses × 36 bits	0	Zero Pad	$A(\rho_{2048}, \theta_{N_\theta - i})$	$A(\rho_{2048}, \theta_i)$
	1	Zero Pad	$A(\rho_{2049}, \theta_{N_\theta - i})$	$A(\rho_{2049}, \theta_i)$
	⋮	⋮	⋮	⋮
	155	Zero Pad	$A(\rho_{2203}, \theta_{N_\theta - i})$	$A(\rho_{2203}, \theta_i)$
	156	Zero Pad	Zero Pad	Zero Pad
	⋮	⋮	⋮	⋮
	511	Zero Pad	Zero Pad	Zero Pad

It is useful to consider the total BRAM consumption at this stage and determine whether further architecture or memory optimisations can be performed. The BRAM allocation for the parallel LHT was previously reported in Table 4.2. For an image of 1920×1080 pixels, the parallel LHT architecture consumed 270 BRAMs, when $\delta_\theta = 1°$, $\delta_\rho = 1$, and θ was across the range $[0, 179]$. An equivalent Symmetric LHT architecture requires 3 BRAMs in total to store the votes for θ_0 and θ_{90}. There are also 89 bit-packed memories that each require 2.5 BRAMs. Therefore, the total BRAM allocation of the bit-packed accumulator is 225.5 BRAM tiles, which is a saving of 44.5 BRAM tiles in comparison to the parallel LHT.

Careful Configuration of BRAM Tiles

Through careful selection of BRAM shapes, it is possible to further reduce the BRAM consumption of the bit-packed accumulator. Consider the example described previously on page 138, where a memory array of 2048 address $\times$ 24 bits was implemented using 1.5 BRAMs. The BRAM configuration for the bit-packed accumulator can be improved using this optimisation technique, as demonstrated in the BRAM allocation table that is presented in Table 5.4. As before, variable i is in the range $[1, 89]$.

Table 5.4: An efficient BRAM allocation table for a memory instance of the bit-packed accumulator when $N_\rho = 2204$, $N_\theta = 180$, and $b = 12$ bits.

BRAM Configuration	**BRAM Address**	**Stored Data (Bits)**			
		[35:27]	[26:24]	[23:12]	[11:0]
36 Kb BRAM: 2048 addresses $\times$ 18 bits and 18 Kb BRAM: 2048 addresses $\times$ 9 bits	0	—	Zero Pad	$A(\rho_0, \theta_{N_\theta - i})$	$A(\rho_0, \theta_i)$
	1	—	Zero Pad	$A(\rho_1, \theta_{N_\theta - i})$	$A(\rho_1, \theta_i)$
	2	—	Zero Pad	$A(\rho_2, \theta_{N_\theta - i})$	$A(\rho_2, \theta_i)$
	3	—	Zero Pad	$A(\rho_3, \theta_{N_\theta - i})$	$A(\rho_3, \theta_i)$
	⋮	⋮	⋮	⋮	⋮
	2047	—	Zero Pad	$A(\rho_{2047}, \theta_{N_\theta - i})$	$A(\rho_{2047}, \theta_i)$
18 Kb BRAM: 512 addresses $\times$ 36 bits	0	Zero Pad	Zero Pad	$A(\rho_{2048}, \theta_{N_\theta - i})$	$A(\rho_{2048}, \theta_i)$
	1	Zero Pad	Zero Pad	$A(\rho_{2049}, \theta_{N_\theta - i})$	$A(\rho_{2049}, \theta_i)$
	⋮	⋮	⋮	⋮	⋮
	155	Zero Pad	Zero Pad	$A(\rho_{2203}, \theta_{N_\theta - i})$	$A(\rho_{2203}, \theta_i)$
	156	Zero Pad	Zero Pad	Zero Pad	Zero Pad
	⋮	⋮	⋮	⋮	⋮
	511	Zero Pad	Zero Pad	Zero Pad	Zero Pad

Each bit-packed memory now requires 2 BRAMs. Therefore, the total BRAM consumption of the bit-packed accumulator is 181 BRAMs. In comparison to the parallel LHT, the bit-packed accumulator reduces BRAM consumption by 89 BRAMs, which is an approximate memory reduction of 33%.

5.2.5 The Voting Scheme

The votes from two symmetric edge pixels can be applied to the bit-packed accumulator at the same time. Each bit-packed memory instance stores votes for the Hough parameters in pairs using $\rho(\theta)$ and $\rho(180° - \theta)$ as a storage pattern. Therefore, when a bit-packed memory location is accessed for accumulation purposes, the stored votes are sliced and votes can be incremented. The new data is then concatenated together and stored back into the original location in the accumulator memory.

The voting strategies for θ_0 and $\theta_{N_\theta/2}$ are different. For θ_0, the accumulator memory will need to be accessed twice to apply votes for each symmetric edge pixel. The accumulator memory for $\theta_{N_\theta/2}$ only needs to be accessed once and may be incremented by two votes, rather than one vote, if each symmetric pixel contains an edge.

5.3 Symmetric LHT Architecture Design

This section describes an FPGA architecture of the Symmetric LHT, which is capable of applying the LHT to an image of 1920 × 1080 pixels using $\delta_\rho = 1$ and $\delta_\theta = 1°$. The operational range of θ is over $[0°, 179°]$. The architecture is capable of applying all possible votes to the HPS. In other words, the HPS was not deliberately reduced in size to optimise memory consumption, or decrease design complexity. The HEP reference design described in Chapter 4 was also used to perform rapid prototyping, hardware validation, and performance analysis of the architecture design.

The remainder of this section will describe the architecture of the Symmetric LHT, which includes its pixel packing system, coordinate calculator, symmetric Hough kernel, accumulator controller, and bit-packed accumulator memory. A system overview of the Symmetric LHT architecture design is presented in Figure 5.6.

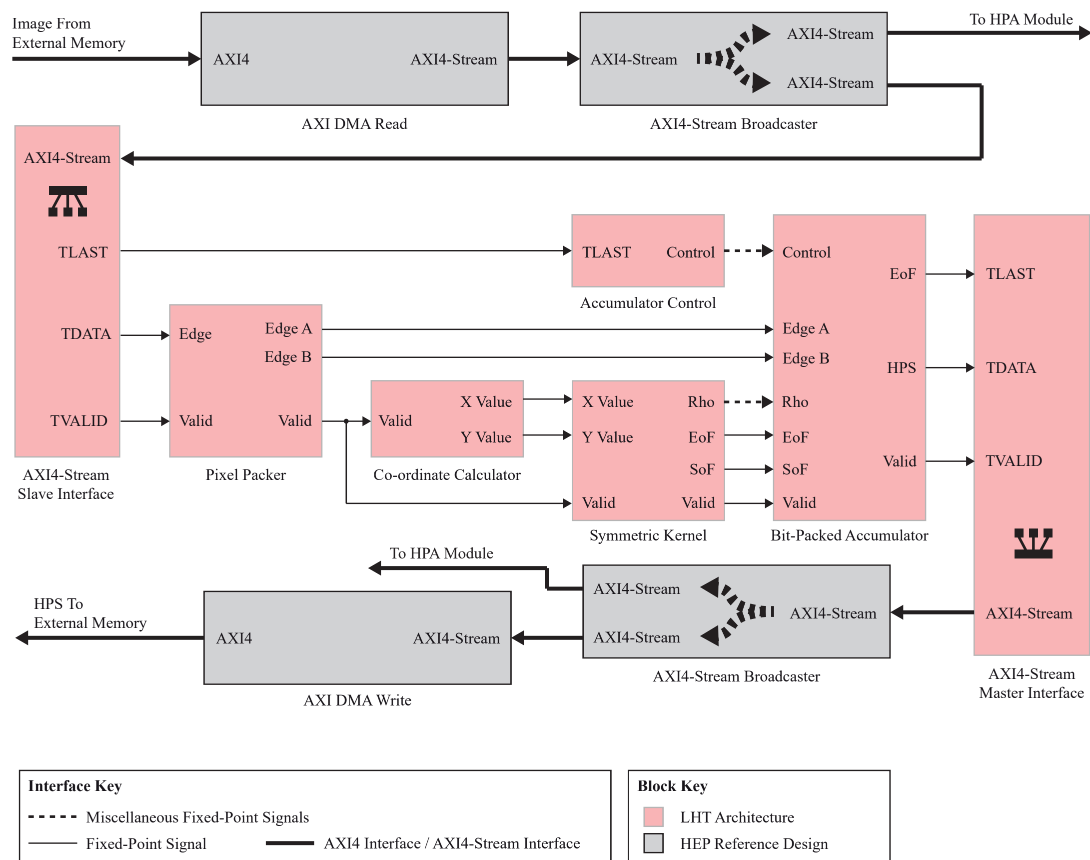


Figure 5.6: System overview of the Symmetric LHT architecture design.

5.3.1 The Pixel Packing System

Image data is typically streamed through the FPGA logic fabric using a raster scan format. However, the Symmetric LHT requires the parallel streaming format described in Section 5.2.3 to operate on two symmetric edge pixels simultaneously. This parallel pixel format streams each image row one after the other, starting from the two centre pixels of a row and moving towards the row's borders.

An FPGA architecture can be designed to change the raster streaming format of the input image to the parallel pixel processing format required by the Symmetric LHT. The architecture design requires a FILO buffer and two FIFO buffers, as illustrated in the pixel packing architecture shown in Figure 5.7.

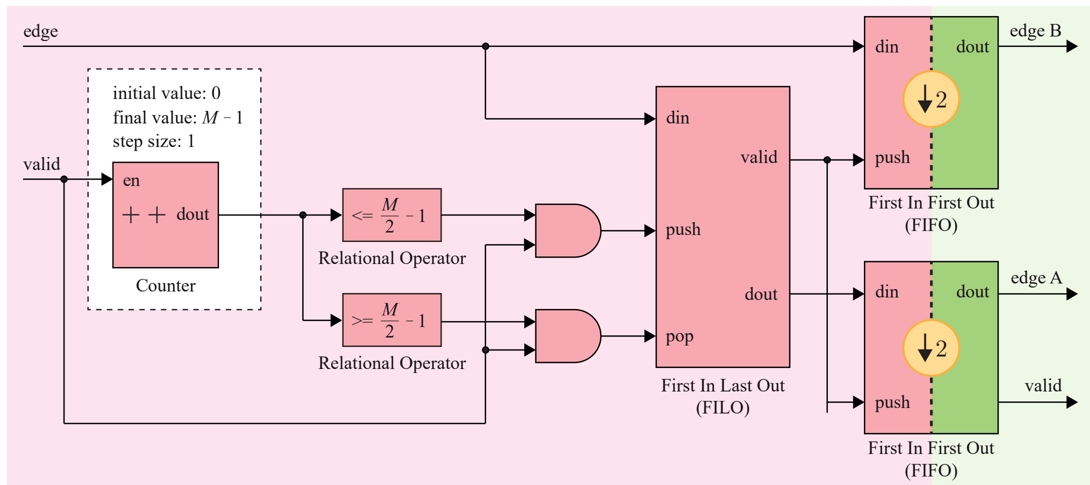


Figure 5.7: The pixel packing architecture that converts from raster streaming format to the parallel pixel streaming format described in Section 5.2.3.

When the edge image is interfaced to the architecture in Figure 5.7, the valid input signal is used to increment a counter to track the x-coordinate of the current image row. If the valid signal is asserted, it indicates the presence of a data payload on the input edge signal. Two relational operators are used to determine whether edge data is pushed into the FILO buffer, or popped from the FILO buffer. Edge data is pushed into the FILO buffer when the first half of the row is streaming into the system. As soon as the second half of the row starts, the FILO is popped.

Each half of the image row is pushed into separate FIFO buffers. Each FIFO pops valid edge data out of the system at half of the input rate. This rate change is possible because the edge data has undergone a serial to parallel conversion, which allows the sample rate to be reduced by two. This sample rate conversion is important later in the symmetric Hough kernel, which employs resource sharing to reduce the consumption of DSP48E2 slices.

Notice that there are three outputs shown in the pixel packing architecture. Starting from the top-right of Figure 5.7, edge B represents the pixels contained in the second half of the input row. Edge A represents the pixels contained in the first half of the input row. These naming conventions are similar to the symmetric pixel example presented previously on the right of Figure 5.4. The valid signal is used to indicate the presence of valid data for the edge A and edge B signals.

5.3.2 Coordinate Calculator

The coordinate calculator maintains the x and y position of the current pixel while the test image is streamed through the Symmetric LHT architecture. The coordinates (x, y) of the current pixel are essential for computing the Hough parameters given by (3.2). The FPGA architecture of the coordinate calculator is a simple design constructed of two fixed-point counters, as shown in Figure 5.8.

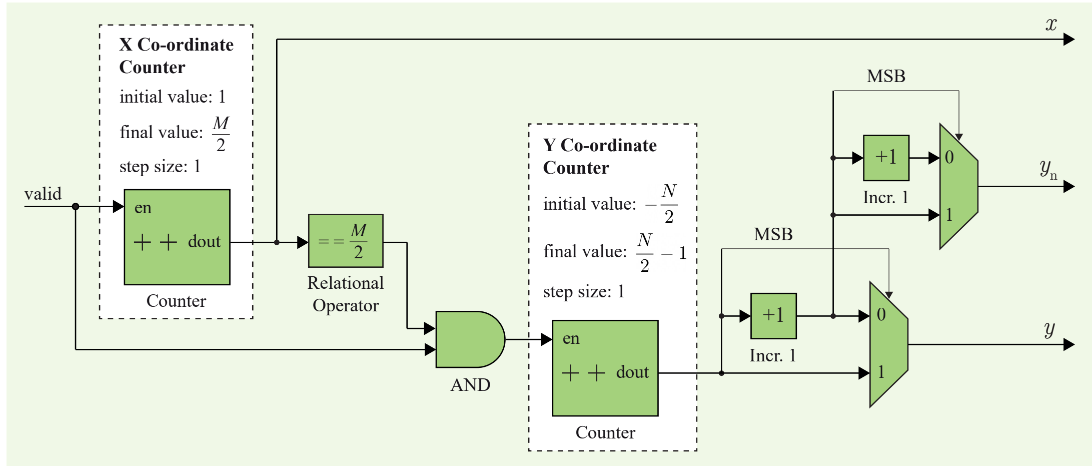


Figure 5.8: FPGA architecture for a simple coordinate calculator. This design uses two fixed-point counters; one counter tracks the x position, and another counter tracks the y position.

The (x, y) values of the current pixel are maintained using one counter for the x position, and another counter for the y position. As previously described in Section 5.2.3, the input image is folded in half across the x-axis. Therefore, the x-coordinate counter only needs to operate across the range $[1, M/2]$. The y-coordinate counter maintains the position for all rows of the input image. Therefore, this counter operates across the range $[-N/2, N/2 - 1]$.

The x counter accumulates by one when the input valid signal is high. A relational operator is used to check if the x counter is equal to $M/2$. The output of the relational operator transitions from low to high when this condition is met, which increments the y counter by one. This mode of operation continues until the x and y counters are equal to $M/2$ and $N/2 - 1$, respectively. When this condition occurs, both counters return to their initial values, ready for the next image.

The coordinate calculator contains three output signals. The first is the x-coordinate value, which is generated using the x counter. The two remaining outputs require additional logic to implement the coordinate system that was previously described in Section 5.2.3. The y-coordinate output is generated using the y counter, which is combined with a multiplexer and an increment-by-one block. This configuration ensures that the y-coordinate output is a non-zero integer, which is required by the Symmetric LHT's coordinate system. The 'next y-coordinate' output represents the next y-coordinate value to occur after the current row is processed. This signal is used later by the Look-Ahead Kernel in Section 5.3.4 to reduce FPGA resource consumption.

5.3.3 Phase Controller

The symmetric Hough kernel can leverage resource sharing techniques since the input image has undergone a serial to parallel conversion as described in Section 5.3.1. To support resource sharing, all input signals to the symmetric Hough kernel and bit-packed accumulator must undergo a sample and hold operation, where the input signal is held for two clock-cycles. Without this conversion, arithmetic and memory resources are unable to reuse values correctly. Additionally, a new signal known as 'phase' is generated using a zero insertion operation on the valid input signal. The phase signal is used to support the arithmetic resource sharing design in Section 5.3.4. The diagram in Figure 5.9 presents the phase controller's architecture.

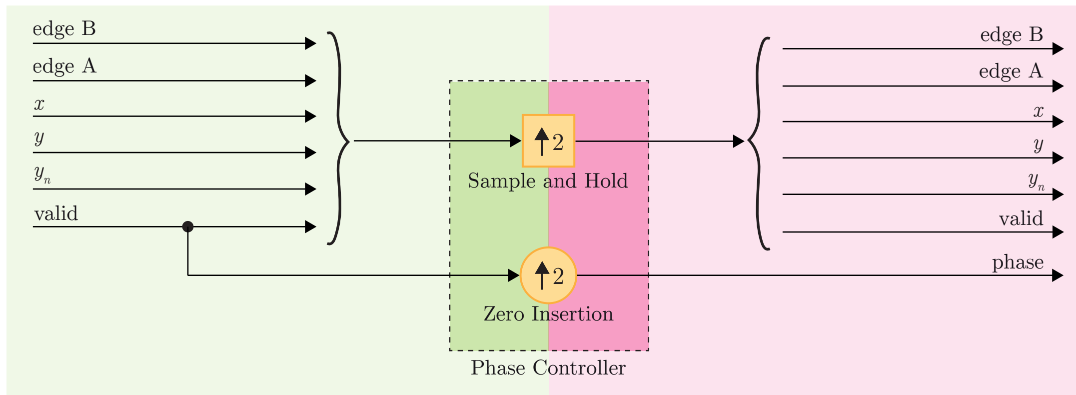


Figure 5.9: A simple phase controller design for the Symmetric LHT. The phase controller generates several signals for resource sharing operations in the symmetric Hough kernel.

5.3.4 Symmetric Hough Kernel

A generalised architecture for the symmetric Hough kernel is presented in Figure 5.10. This architecture implements (3.2), while using the relationship in (5.1) to reduce FPGA arithmetic resource consumption.

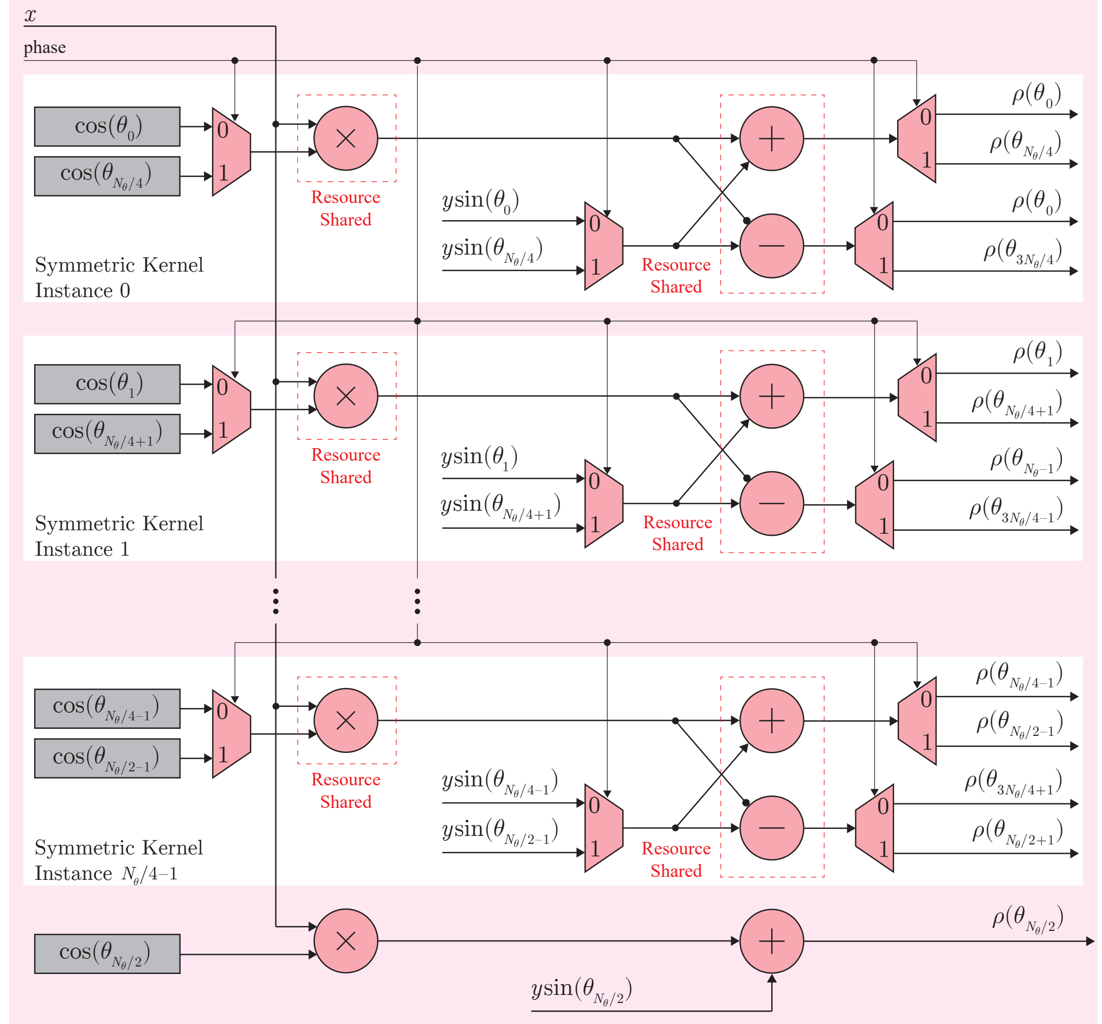


Figure 5.10: Generalised architecture for the symmetric Hough kernel.

The symmetric kernel architecture demonstrates that four values in $\rho(\theta)$ can be computed using only one multiplier and two adders. The multipliers are used to compute $x\cos(\theta)$, where each value in $\cos(\theta)$ is pre-calculated and stored in ROM. The phase input allows the architecture to swap between different values of $\cos(\theta)$ and $y\sin(\theta)$ during operation.

It is possible to optimise the symmetric kernel architecture in Figure 5.10. Multiplier resources can be reduced through careful selection of θ values. For example, if $\theta_0 = 0°$ and $\theta_{N_\theta/2} = 90°$, then $x\cos(0°) = x$ and $x\cos(90°) = 0$, which does not require multipliers. Furthermore, if at least one value in θ is equal to 60°, then $x\cos(60°) = x/2$, which can be implemented using a simple bit-shift operation.

The symmetric kernel architecture must be appropriately constrained to prevent the overallocation of FPGA resources. It is necessary to ensure that N_θ is an even number, and the values in θ must also be evenly spaced using a regular discrete step across the range $[0°, 180°)$. The wordlength of the multipliers should be carefully selected to prevent unnecessary allocation of DSP48E2 slices. The symmetric kernel architecture in this thesis sets the wordlength of x to $\lceil \log_2(M) \rceil$, and the wordlength of $\cos(\theta)$ to 16 bits, where 14 bits are used for fractional representation. The adder's accumulator uses full precision, the output is rounded, and the output wordlength is set to $\lceil \log_2(D) \rceil$, where D is the length of the image diagonal given by (3.4).

The DSP48E2 slice consumption for the parallel LHT was previously reported in Table 4.2. For an image of 1920×1080 pixels, the parallel LHT architecture consumed 89 DSP48E2 slices, when $\delta_\theta = 1°$, $\delta_\rho = 1$, and θ was across the range [0, 179]. In particular, 88 DSP48E2 slices were used to calculate $x\cos(\theta)$, while 1 DSP48E2 slice computes $y\sin(\theta)$ for each image row. An equivalent Symmetric LHT architecture only requires 45 DSP48E2 slices to implement the symmetric Hough kernel. The $x\cos(\theta)$ multiplication is implemented using 44 DSP48E2 slices, while $y\sin(\theta)$ is calculated using 1 DSP48E2 slice.

The Look-Ahead Hough Kernel

Notably, the symmetric Hough kernel presented in Figure 5.10 contains several input signals named $y\sin(\theta_i)$. These signals originate from the Look-Ahead Kernel, which was described in [20] to reduce FPGA resource consumption. The Look-Ahead Kernel calculates $y\sin(\theta)$ for the next image row and stores the results in output registers. This architecture was used to reduce DSP48E2 slice consumption for the symmetric Hough kernel, and was previously illustrated in Figure 3.19 on page 75.

The Look-Ahead Kernel uses a fixed-point counter and a LUT to cycle through values of $\sin(\theta)$. The $\sin(\theta_i)$ values are multiplied with the y value for the next row, denoted as y_n. The multiplier output, $y_n \sin(\theta_i)$, is sent to a tapped register bank, which is connected to a set of output registers. There are N_θ output registers that are only enabled when the current row is ending, which allows new values of $y_n \sin(\theta)$ to be loaded for the next row. When the first row of an image is about to be processed, the output registers are reset to an initial value of $-(N/2)\sin(\theta)$, allowing the architecture to be easily primed for the next image.

5.3.5 Accumulator Controller

The accumulation array is an autonomous circuit that requires a controller to maintain correct operation. The controller is implemented using a Finite State Machine (FSM) designed as a Moore machine. Figure 5.11 presents an illustration of the top-level accumulation architecture which includes an accumulator controller, accumulator memory, and an angle correction and selection subsystem.

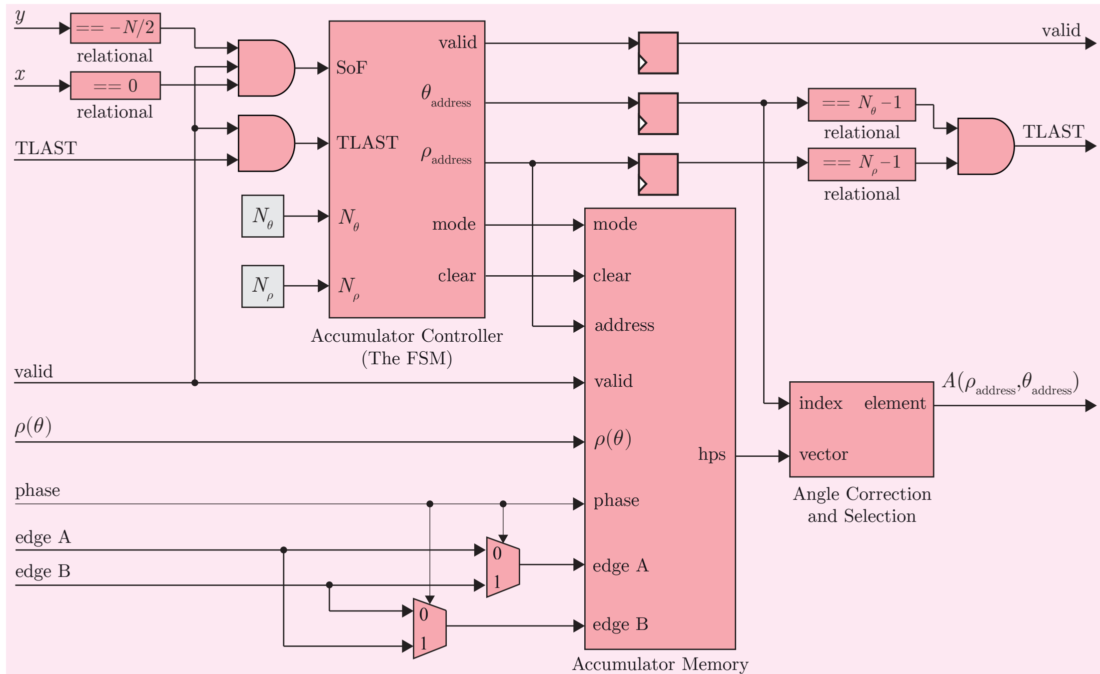


Figure 5.11: FPGA architecture for the top-level accumulator design. The accumulation controller implements an FSM that maintains the operation of the accumulation memory. Control logic has been removed from this design to simplify the illustration.

The x and y inputs are used to create a Start of Frame (SoF) signal using relational operators and a logical AND gate. The SoF signal is used to inform the accumulator controller that a new image is currently streaming into the architecture. The input signals to the FSM are the SoF signal, the TLAST signal from the read DMA, and two constants, N_ρ and N_θ. The FSM also contains three internal registers: the Current State register, a ρ counter register denoted as ρ_{count}, and a θ counter register denoted as θ_{count}. Both the ρ_{count} and θ_{count} registers are connected to the output of fixed point counters, which accumulate during operation.

The FSM has four states of operation: IDLE, VOTE, READ, and CLEAR. The FSM diagram, which contains the controller states and transitions, can be seen in Figure 5.12. The default entry state that initialises the FSM is the IDLE state, which is annotated on the diagram as 'initialise'.

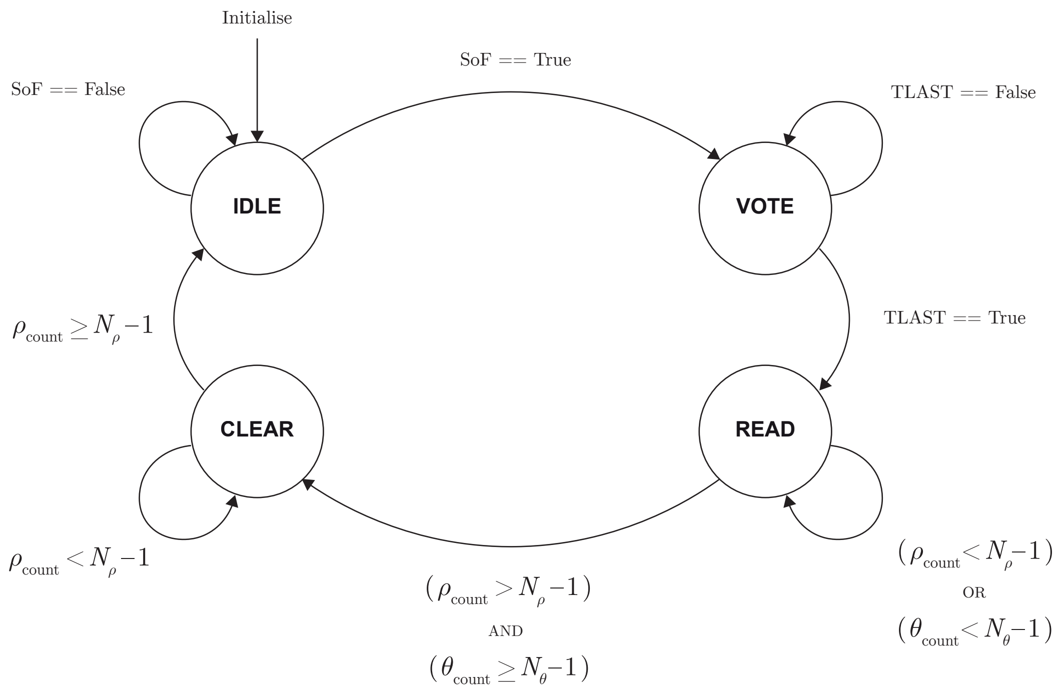


Figure 5.12: FSM diagram for the accumulator controller. The FSM contains four states: IDLE, VOTE, READ, and CLEAR. The default entry state is IDLE. If an undefined state occurs, the FSM returns to the IDLE state.

In the IDLE state, the FSM waits for the SoF signal to assert. When this occurs, the FSM transitions to the VOTE state, where votes are accumulated in memory until the TLAST signal is asserted. The TLAST signal informs the controller that the last pixel of the image has been processed. The FSM then transitions into the READ state, where the contents of the accumulator memory are read by generating addresses using the ρ and θ counters. When all addresses have been accessed, the FSM transitions to the CLEAR state. The accumulator is reset by simply cycling through all ρ addresses and zeroising each memory location. When the clear operation is complete, the FSM returns to the IDLE state and waits for the next image.

The FSM contains five output signals: mode, clear, valid, ρ_{address}, and θ_{address}. The 'mode' output is used by the accumulator memory to switch voting on and off. The 'clear' signal is used to set the data stored in the current memory location to a value of zero. The 'valid' output is used to inform the write DMA of valid HPS data. Finally, the ρ_{address} and θ_{address} outputs are used to address accumulator memory locations and index individual memories.

The outputs for each FSM state are shown in Table 5.5. The mode, clear, and valid outputs are each set to either True or False, depending on the given state. The ρ_{address} and θ_{address} outputs are always set to the value of the ρ_{count} and θ_{count}, respectively.

Table 5.5: Accumulator controller output values.

Output	**IDLE**	**VOTE**	**READ**	**CLEAR**
mode	False	True	False	False
valid	False	False	True	False
clear	False	False	False	True
$\rho_{\textbf{address}}$	ρ_{count}	ρ_{count}	ρ_{count}	ρ_{count}
$\theta_{\textbf{address}}$	θ_{count}	θ_{count}	θ_{count}	θ_{count}

The value of the Current State register and the conditions required to influence the next state can be seen in Table 5.6. As shown, when the Current State register is IDLE, then the next possible state is VOTE. The next state is selected using the condition column i.e. SoF == True. In this example, if the SoF input remains False, then the condition is not met and the Current State register remains in the IDLE state.

Table 5.6: Accumulator controller configuration for the Current State register.

FSM Register	Value	Next Value	Condition
Current State	IDLE	VOTE	SoF == True
	VOTE	READ	TLAST == True
	READ	CLEAR	($\rho_{\text{count}} \geq N_\rho - 1$) and ($\theta_{\text{count}} \geq N_\theta - 1$)
	CLEAR	IDLE	$\rho_{\text{count}} \geq N_\rho - 1$

Note that Table 5.6 is another way of representing the FSM diagram given in Figure 5.12. This table is a useful and clear way of quickly identifying the conditions required for registers to change value. Table 5.7 presents a similar table to represent the ρ_{count} and θ_{count} registers. As shown, the ρ_{count} and θ_{count} registers are not modified in the IDLE or VOTE states. In the READ state, the ρ_{count} register is incremented by one after each clock cycle until $\rho_{\text{count}} \geq N_\rho - 1$. When this condition is met, the θ_{count} register is incremented by one and the ρ_{count} register is set to 0. This process repeats until both $\theta_{\text{count}} \geq N_\theta - 1$ and $\rho_{\text{count}} \geq N_\rho - 1$. When this condition occurs, both registers are set to zero and the Current State register transitions to the CLEAR state as given in Table 5.6.

Table 5.7: Accumulator controller configuration for the ρ_{count} and θ_{count} registers. A 'No Condition' entry in the condition column indicates that the transition between the current value and the next value always occurs.

FSM Register	State	Value	Next Value	Condition
ρ_{count}	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	READ	0	1	$\rho_{\text{count}} < N_\rho - 1$
		1	2	$\rho_{\text{count}} < N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\rho - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
	CLEAR	0	1	$\rho_{\text{count}} < N_\rho - 1$
		1	2	$\rho_{\text{count}} < N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\rho - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
θ_{count}	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	READ	0	1	$\rho_{\text{count}} \geq N_\rho - 1$
		1	2	$\rho_{\text{count}} \geq N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\theta - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
	CLEAR	0	0	No Condition

The READ state allows the accumulator controller to cycle through all addresses of ρ in the range $[0, N_\rho - 1]$. The θ_{count} register is incremented when $\rho_{\text{count}} \geq N_\rho - 1$, as this allows all θ values to be read from the accumulator memory (described in Section 5.3.6). The CLEAR state reuses the ρ_{count} register to cycle through all addresses of the accumulator memory and set the corresponding memory locations to zero. The θ_{count} register is not required as it is not necessary to index individual θ locations when clearing the accumulator memory.

5.3.6 Accumulator Circuit Design

The contents of the accumulator memory subsystem will now be presented. Figure 5.13 provides a diagram of the accumulator memory subsystem, which contains three unique memory blocks for storing and accumulating votes. The subsystems labelled 'θ_0 Accumulator' and '$\theta_{N/2}$ Accumulator' store votes that belong to θ_0 and $\theta_{N/2}$, respectively. The subsystem named 'Bit-Packed Accumulator' stores votes for $\theta_1, \theta_2 ... \theta_{N_\theta/2-1}$ and $\theta_{N_\theta-1}, \theta_{N_\theta-2} ... \theta_{N_\theta/2+1}$. The phase input is only used by the 'θ_0 Accumulator' to perform a unary minus operation on the $\rho(\theta_0)$ input.

Bit-Packed Accumulator Circuit Design

The architecture design for one instance of the bit-packed accumulator can be seen in Figure 5.14. Note that BRAM port abbreviations and modes were previously covered in Section 2.3.3. The BRAM is configured in SDP mode, which allows one read and write transaction to occur per clock cycle. Votes are accumulated by first accessing a location in memory using the 'rdaddr' port. After one clock cycle, the data stored at the memory location will be presented at the 'dout' port. The data is sent through the accumulation path, where it is appropriately sliced to increment each location in the HPS by one vote if an edge pixel exists at the corresponding location in the input image. After voting, the data is concatenated and sent to the BRAM's 'din' port. The 'wraddr' port writes the data back into the original address location if the 'wren' port on the BRAM is a logical '1'.

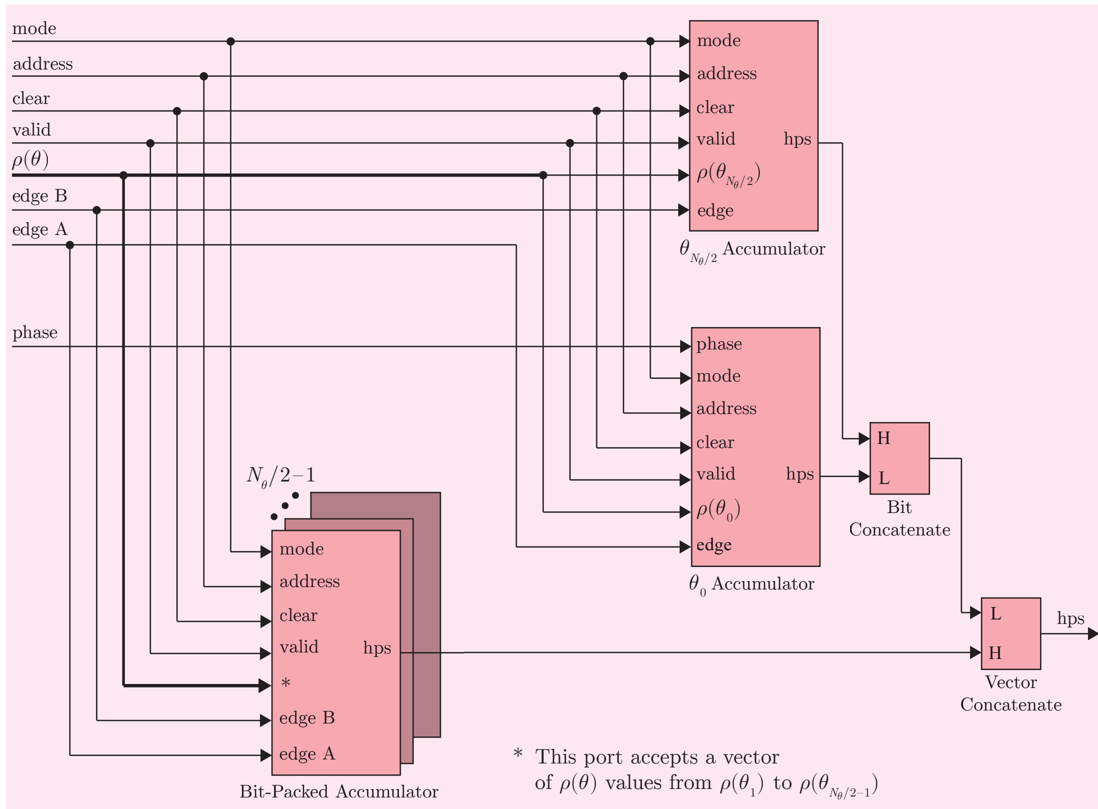


Figure 5.13: Overview diagram of the accumulator memory subsystem. Notice that there are $N_\theta/2 - 1$ bit-packed accumulator memories.

If read and write operations occur at the same address, then a BRAM operating in SDP mode will present old data at the 'dout' port (instead of the new data to be written). For this reason, votes can be dropped if two or more consecutive edge pixels share the same address in memory. To correct this issue, there is additional control logic below the BRAM shown in Figure 5.14. A relational operator monitors the BRAM's read and write address ports to check if they are equal. When the read and write addresses are the same, a logical AND gate checks if the addresses correspond to a voting edge pixel. If true, then the multiplexer on the right of Figure 5.14 will loop data that was incremented in the previous clock cycle back into the accumulation path. When the read and write address ports are no longer equal to one another, the accumulated data is written back into its corresponding location in the accumulator memory. Notice that the accumulation design uses the value of the associated edge pixel to increment a location in memory by one.

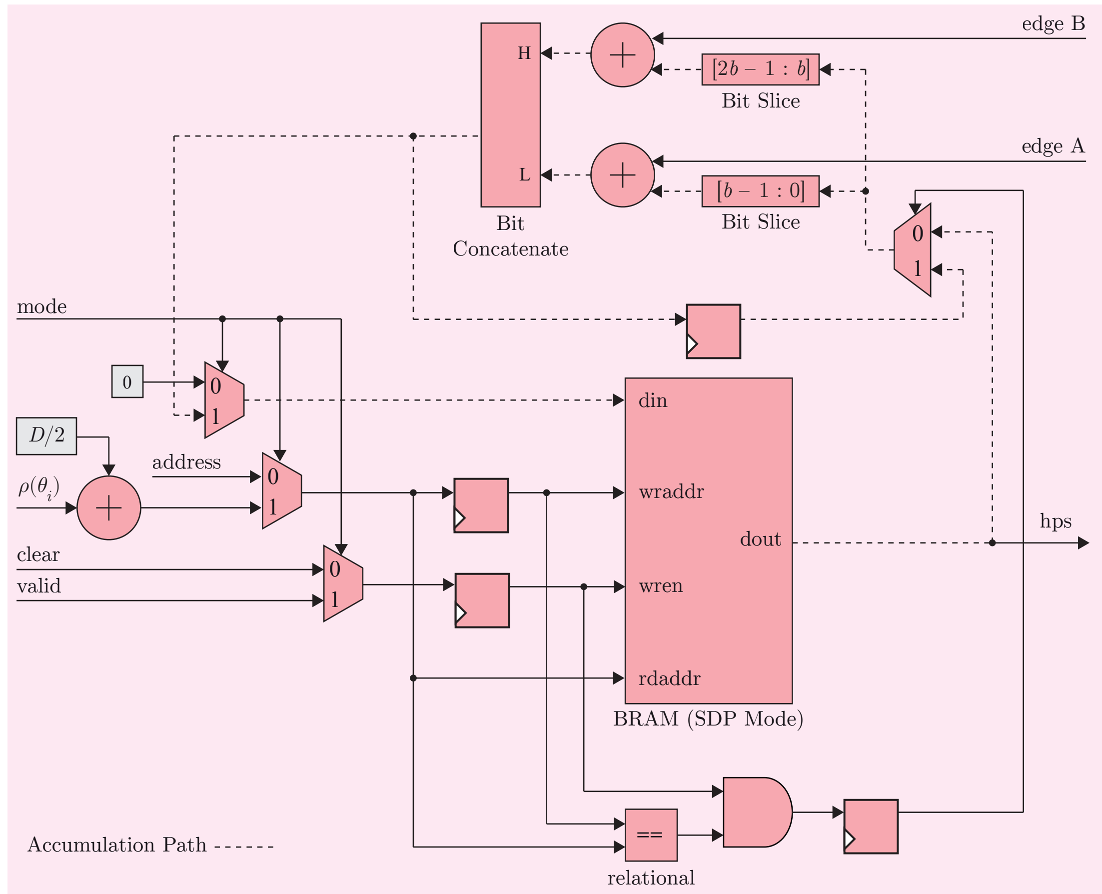


Figure 5.14: The bit-packed accumulator memory architecture design that performs voting for two angles in θ. The BRAM selected is configured in SDP mode, which allows one read and write transaction per clock cycle. The accumulation path is indicated by the dashed line.

The mode input allows the accumulator controller to switch voting operations on and off. When mode is set to logical '1', the multiplexers in Figure 5.14 are configured to apply votes to the BRAM. When mode is set to logical '0', the accumulator controller can read all locations in the accumulator memory using the ρ_{address} input. The clear input may also be used to erase the accumulator memory for the next image.

When the Hough parameters are calculated using the symmetric Hough kernel, their range is between $[-D/2, D/2]$. The accumulator memory can only be indexed using whole numbers. Therefore, it is necessary to shift the range of $\rho(\theta)$ such that it resides in the range $[0, D]$. This operation is achieved on the left of Figure 5.14 using an adder, where $\rho(\theta_i) + D/2$ is performed (i is a variable in the range $[1, 89]$).

θ_0 and $\theta_{N_\theta/2}$ Accumulators

Figure 5.15 presents a general circuit diagram for the θ_0 and $\theta_{N_\theta/2}$ accumulators. This architecture can be used for both accumulators. The θ_0 accumulator is unique because it uses the phase input to perform a unary minus operation on the input address.

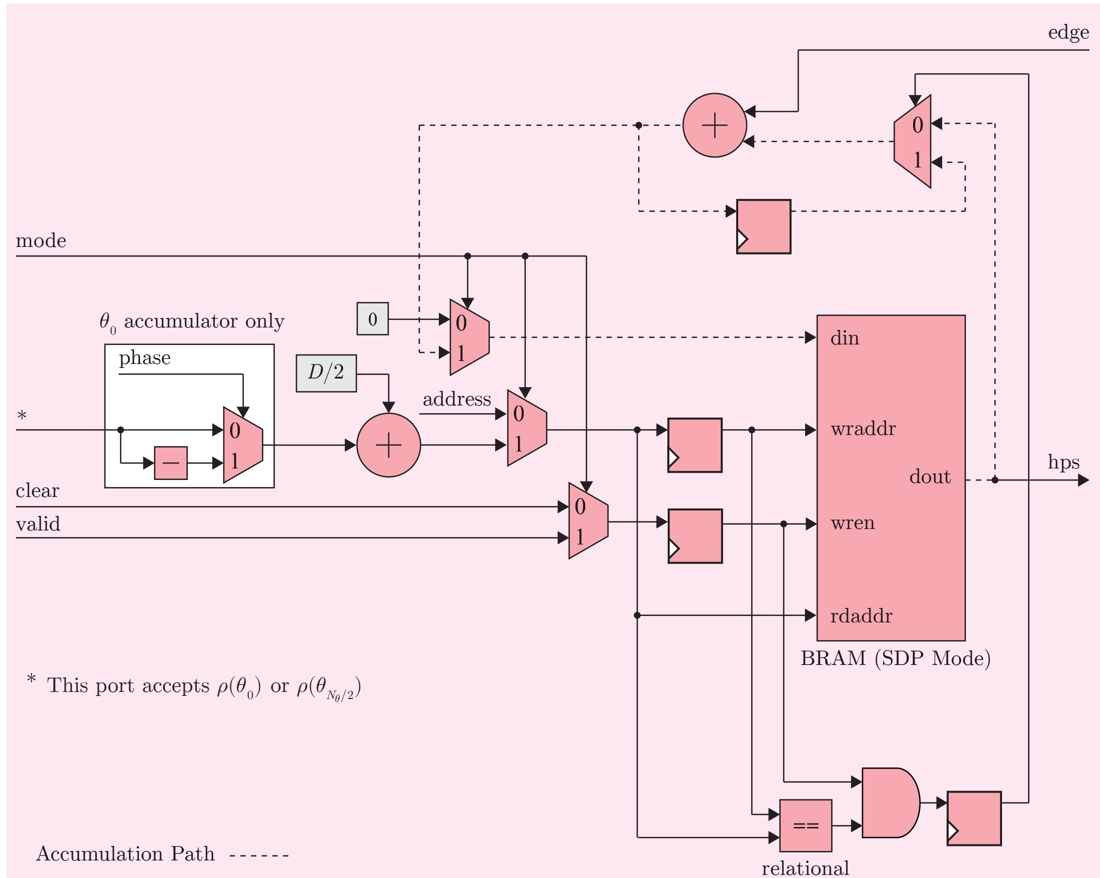


Figure 5.15: A general circuit diagram for the θ_0 and $\theta_{N_\theta}/2$ Accumulators. The accumulation path is indicated by the dashed line.

The θ_0 Accumulator only uses one adder to increment votes and applies a unary minus operation to the input address if phase = 1. The reason for applying a unary minus operation can be explained by considering the symmetric points A and B from Figure 5.4, on page 134. Since $\rho_A(\theta_0) = -\rho_B(\theta_0)$, then a unary minus can be used to negate the value of the input address to the θ_0 accumulator. In contrast, the $\theta_{N_\theta/2}$ Accumulator does not require a unary minus operation since $\rho_A(\theta_{N_\theta/2}) = \rho_B(\theta_{N_\theta/2})$.

5.4 Analysis and Evaluation

This section evaluates the Symmetric LHT architecture for its FPGA resource consumption and timing performance. The implementation results are compared with LHT architectures from previously published works to highlight the efficiency of the Symmetric LHT design. Details on how to inspect the Symmetric LHT electronically are provided in [106].

5.4.1 Implementation Results

The Vivado Design Suite was used to synthesise and implement the Symmetric LHT architecture to evaluate its FPGA resource consumption and timing performance. The Symmetric LHT is implemented alongside the HEP for visualisation of the HPS and analysis of the architecture design. The maximum clock frequency of the Symmetric LHT architecture was reported to be 205 MHz. The architecture design was configured to apply the LHT to an image of 1920×1080 pixels and the discrete step of the HPS was set to $\delta_\rho = 1$ and $\delta_\theta = 1°$. The FPGA resource allocation of the Symmetric LHT architecture was reported for the XCZU7EV-2E device as shown in Table 5.8.

Table 5.8: FPGA resource consumption for the Symmetric LHT architecture (without the HEP), which is implemented on the XCZU7EV-2E device.

Resource	Available	Used	Percentage (%)
LUTs	230,400	15,529	6.74
LUT RAM	101,760	66	0.06
FFs	460,800	10,065	2.18
BRAM	312	181	58.01
DSP48E2	1,728	45	2.60

The symmetric Hough kernel uses 45 DSP48E2 slices to implement (3.2). The bit-packed accumulator memory requires 181 BRAM tiles to store the HPS. Each accumulator subsystem uses two 36 Kb BRAM tiles to store votes for two angles in θ. The exceptions to this are the accumulator subsystems used to store the first and centre angles in θ, which require one 36 Kb BRAM and one 18 Kb BRAM to store votes. Note that the parallel LHT architecture given in Appendix B uses 270 BRAMs to store the HPS. The Symmetric LHT has decreased BRAM tile consumption by 32.96%.

The parallel LHT detailed in [87] and the Symmetric LHT described in this chapter were each implemented on the XCZU7EV-2E device for a range of image resolutions commonly found in literature. These architectures were designed using the HEP and the HPS was configured for $\delta_\rho = 1$, $\delta_\theta = 1^\circ$, and $N_\theta = 180$. Table 5.9 presents the memory requirements for each set of architectures, which can also be investigated in [106]. Additionally, all Symmetric LHT architectures require 45 DSP48E2 slices, while the parallel LHT architectures use 89 DSP48E2 slices.

Table 5.9: The memory consumption of the parallel LHT and the Symmetric LHT for various image resolutions.

Resolution (Pixels)	Parallel LHT			Symmetric LHT			Memory Saved(%)
	BRAM		Total Bits	BRAM		Total Bits	
	18 Kb	36 Kb		18 Kb	36 Kb		
320×240	180	—	3,317,760	91	—	1,677,312	49.44
333×333	180	—	3,317,760	91	—	1,677,312	49.44
512×512	180	—	3,317,760	2	89	3,317,760	0.00
800×600	180	—	3,317,760	2	89	3,317,760	0.00
1024×768	—	180	6,635,520	89	91	4,995,072	24.72
1280×720	—	180	6,635,520	89	91	4,995,072	24.72
1920×1080	180	180	9,953,280	180	91	6,672,384	32.96

The final column of Table 5.9 indicates the memory saved by the Symmetric LHT compared to the parallel LHT. Notably, the two smallest image resolutions exhibit the greatest memory savings. The parallel LHT inefficiently allocates 180 18 Kb BRAMs for each angle in θ. The Symmetric LHT can efficiently bit-pack the votes for two angles in θ in the same memory location, requiring 91 18 Kb BRAMs.

The Symmetric LHT does not exhibit memory savings for the image resolutions 512×512 pixels and 640×480 pixels. The bit-packed accumulator for each resolution uses $b = 10$ bits and requires $N_\rho = 800$ and $N_\rho = 1000$, respectively. The BRAM configuration for the bit-packed accumulator is 1024 addresses $\times$ 36 bits, which requires one 36 Kb BRAM for storing the votes of two angles in θ. A parallel LHT architecture for each image resolution uses an 18Kb BRAM configuration of 1024 addresses $\times$ 18 bits for one angle in θ, which is equivalent in memory consumption to the Symmetric LHT. Memory requirements cannot be reduced by the Symmetric LHT for these resolutions as the available BRAM schemes cannot effectively store the bit-packed accumulator.

The remaining two image resolutions are 1024×768 pixels and 1280×720 pixels. The bit-packed accumulator for each of these resolutions stores the votes for two angles in θ using one 36 Kb BRAM and one 18 Kb BRAM. An equivalent parallel LHT architecture requires one 36 Kb BRAM to store the votes for one angle in θ, which is less memory-efficient. In Section 5.4.4, the memory requirements of the Symmetric LHT are compared to previously published FPGA implementations of the LHT.

5.4.2 Processing Time Results

Previously in Section 4.2.3, the HEP accurately calculated the processing time of the parallel LHT architecture. The processing time of the Symmetric LHT architecture for a 1920×1080 pixel image is reported by the HEP as 12.06 ms. The frame rate of this Symmetric LHT architecture corresponds to approximately 82.95 fps, which is able to process the FHD video standard.

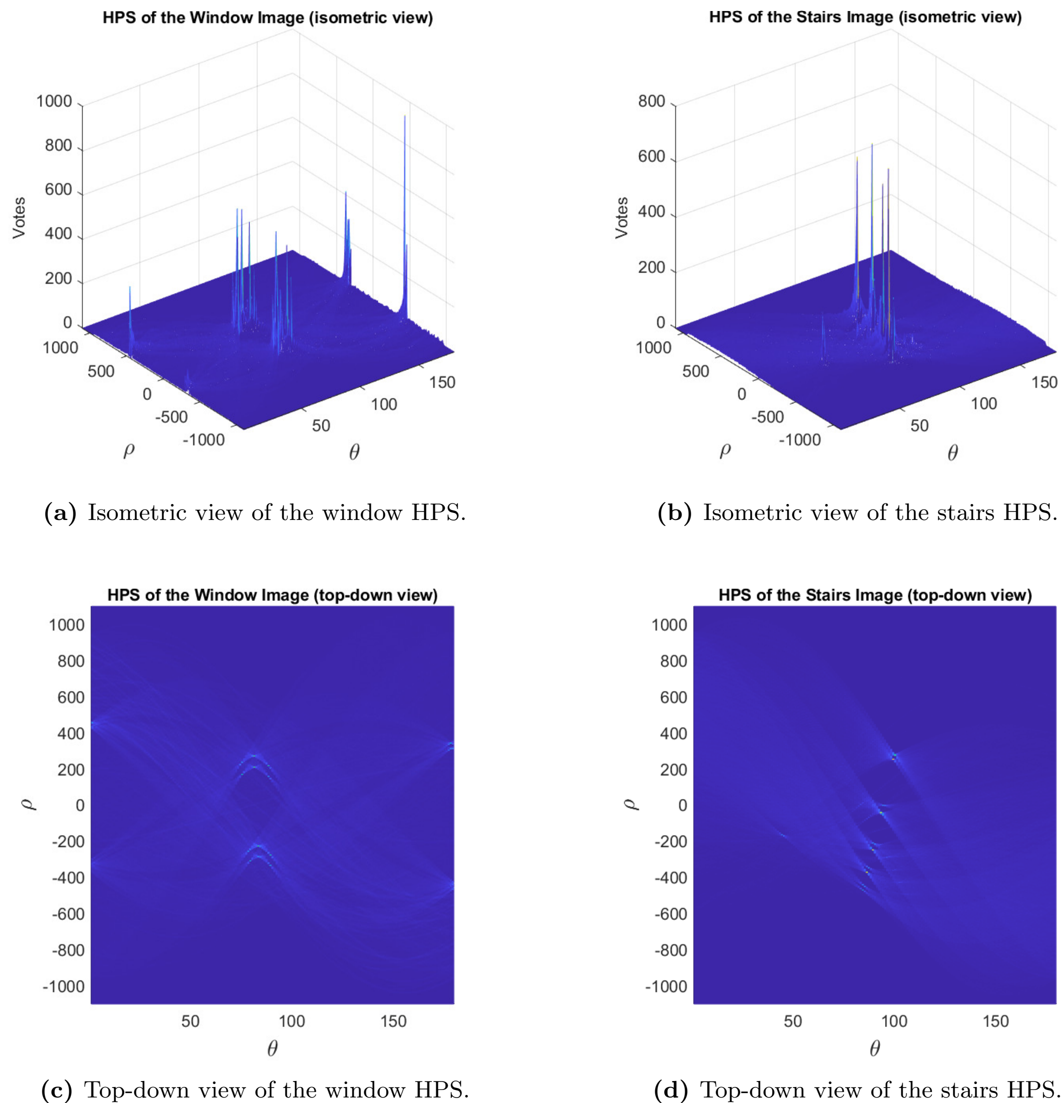
The processing time analysis using the HEP can be performed for Symmetric LHT architectures that target other image resolutions, as shown in Table 5.10. Notice that most architectures achieve a target clock frequency of 250 MHz, while the resolutions corresponding to 1024×768 pixels and 1280×720 pixels achieve a maximum clock frequency of 235 MHz. Larger image resolutions require more BRAM to store the HPS, which increases routing delays resulting in longer critical paths, limiting the maximum achievable clock frequency of the architecture. Notice that the smallest resolution, 320×240 pixels, achieves the fastest frame processing time of 0.60 ms, which corresponds to 1,666.67 fps. Each value in Table 5.10 is rounded to two decimal places.

Table 5.10: Processing time results of Symmetric LHT architectures that target various image resolutions. The Processing Time column contains measurement results from the HEP.

Resolution (Pixels)	**Clock Frequency (MHz)**	**Processing Time (ms)**	**Frames Per Second (fps)**
320×240	250.00	0.60	1,666.67
333×333	250.00	0.79	1,265.82
512×512	250.00	1.57	636.94
800×600	250.00	2.64	378.79
1024×768	235.00	4.33	230.95
1280×720	235.00	5.05	198.02
1920×1080	205.00	12.06	82.92

5.4.3 Architecture Validation and Testing

The validation procedure for testing FPGA architectures of the LHT was previously described in Section 4.4.3. The Symmetric LHT architecture design was successfully hardware validated using edge images of the window and stairs, which are given in Figure 4.14 on page 120. The HPS returned by the Symmetric LHT architecture for each test image are presented in Figure 5.16.



(a) Isometric view of the window HPS.

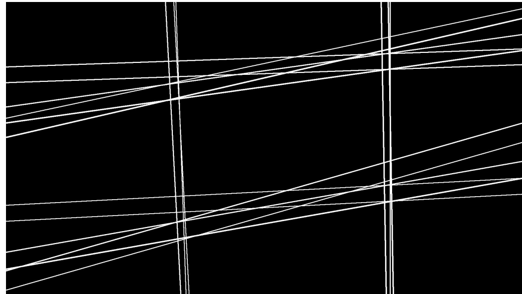
(b) Isometric view of the stairs HPS.

(c) Top-down view of the window HPS.

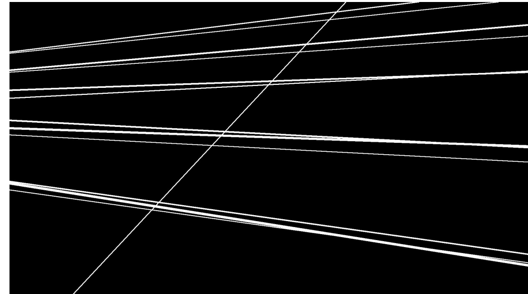
(d) Top-down view of the stairs HPS.

Figure 5.16: HPS results for the hardware validation of the Symmetric LHT architecture on the XCZU7EV-2E device. The isometric view of the HPS for the window image (a), and the stairs image (b). The top-down view of the HPS for the window image (c), and the stairs image (d).

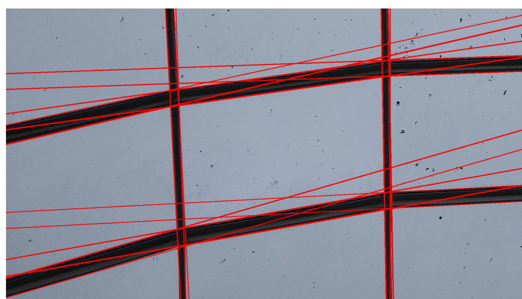
The parameters of the peaks detected in the HPS for each test image are used in (3.6) and (3.7) to reconstruct lines. The line reconstruction results for the Symmetric LHT are presented in Figure 5.17.



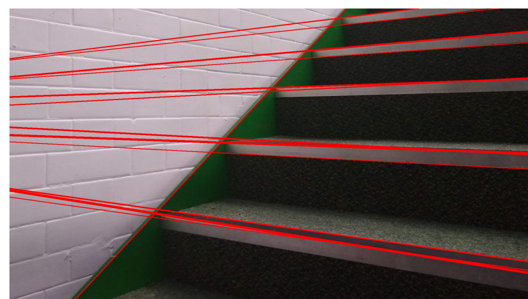
(a) Reconstructed line image of the input window edge image.



(b) Reconstructed line image of the input stairs edge image.



(c) Overlay of the reconstructed image and the original colour image of the window.



(d) Overlay of the reconstructed image and the original colour image of the stairs.

Figure 5.17: Line reconstruction results for the test images input into the Symmetric LHT architecture. The reconstructed line images of the window (a) and stairs (b). The reconstructed lines overlaid on top of the original colour images of the window (c) and stairs (d).

These results indicate that the Symmetric LHT has the same line detection capabilities as the parallel LHT, while consuming fewer FPGA arithmetic and memory resources. The line reconstruction images for the parallel LHT in Figure 4.16 and the Symmetric LHT in Figure 5.17 both return the same images. This equivalence check was performed by iterating through the elements of both images and using the equality operator to determine if both images are equal. The Symmetric LHT was also validated on hardware using the Jupyter Lab environment. Screenshots of the validation notebooks are provided in Appendix C.

5.4.4 Comparison with Previously Published Work

This section compares the Symmetric LHT with previously published implementations of the LHT, with architectures compared based on their memory consumption. The Symmetric LHT architecture is set to have the same configuration as other LHT architectures where possible. For instance, if an LHT architecture uses an image resolution of 800×600 pixels and $\delta_\theta = 11.25°$, then the Symmetric LHT is set to the same configuration to enable the direct comparison of memory requirements. See [106] for practical implementations of all custom Symmetric LHT architectures described in this section.

Zhou *et al.* [20] implemented a gradient-based parallel LHT that requires 6,635,520 bits of memory for a 333×333 pixel image. For the same image resolution, the Symmetric LHT uses 1,677,312 bits, saving 74.72% of memory. The same authors developed a low-latency parallel LHT architecture in [14] that consumed 3,317,760 bits of dedicated FPGA memory for an image of 512×512 pixels. The Symmetric LHT uses an equivalent amount of memory when processing the same resolution.

Chen *et al.* [89] leveraged external memory in their LHT architecture to store the HPS. Their work required 223,360 bits of on-chip memory and used off-chip memory to store the HPS and edge image. Their architecture can only process 512×512 pixel images, and the speed and bandwidth of the external memory limit the execution time. Notably, their LHT architecture design sets the number of bits for voting in the HPS to $b = 9$. An equivalent Symmetric LHT architecture uses 1,677,312 bits. The Symmetric LHT uses more on-chip memory than that published in [89]. However, it provides the flexibility for parametrising the architecture for different resolutions with no external memory limitations, which enables low-latency, deterministic processing capabilities.

Elhossini *et al.* [88] created a low-memory architecture that can extract lines and circles from digital images using the LHT and CHT, respectively. Their parameter space architecture can process images of 800×600 pixels and is coarsely discretised such that $\delta_\theta = 11.25°$. The memory consumption of their implementation is 262,144 bits. The Symmetric LHT was generated using the same configuration and consumed 165,888 bits of memory to store the HPS, saving 36.72% of memory.

Bailey in [21] describes an LHT architecture that can process 1024×768 pixel images

using a reduced size HPS, which is configured using $N_\rho = 1024$ and $b = 9$. As a result, Bailey acknowledges that this architecture cannot detect lines near the corners of an image. It is worth noting that Bailey describes a novel technique for efficiently drawing detected lines, which motivated these design decisions. For a resolution of 1024×768 pixels, Bailey's architecture consumes 1,843,200 bits while an equivalent Symmetric LHT architecture (without line drawing functionality) requires 1,677,312 bits of memory to store the HPS, saving 9% of memory.

Lu *et al.* [13] designed an unrolled LHT architecture for an image of 1024×768 pixels, where the HPS was configured to use $N_\theta = 101$ angles. Their architecture requires 3,474,432 bits of dedicated FPGA memory. To compare memory consumption, the Symmetric LHT architecture was generated using $N_\theta = 102$ angles (as the number of angles must be even). While offering a finer δ_θ, the Symmetric LHT consumes 2,838,528 bits, saving memory requirements by 18.30%.

Solod *et al.* [90] uses HLS to generate an LHT architecture for lane detection in vehicles. Their LHT architecture can process an image of 1920×1080 pixels and the HPS is configured using $N_\rho = 2048$ and $N_\theta = 41$. This configuration requires 1,695,744 bits of memory to store the HPS. An equivalent Symmetric LHT architecture that configures the HPS using $N_\theta = 42$ only consumes 1,216,512 bits, which saves 28.26% of memory and offers a finer δ_θ.

Table 5.11 presents a summary for each of the above architecture comparisons. Included in this table is the DSP slice consumption and operational clock frequency of each architecture design.

Table 5.11: Results of the Symmetric LHT and comparison with related works.

Resolution (Pixels)	Related Works				Symmetric LHT		
	Ref.	**Memory (bits)**	**DSP Slices**	**Freq. (MHz)**	**Memory (bits)**	**DSP Slices**	**Freq. (MHz)**
333×333	[20]	6,635,520	13	260.06	1,677,312	45	250.00
512×512	[87]	3,317,760	90	247.53	3,317,760	45	250.00
	[89]	223,360	0	200.00	1,677,312	45	250.00
800×600	[88]	262,144	—	—	165,888	5	250.00
1024×768	[21]	1,843,200	0	73.50	1,677,312	45	250.00
	[13]	3,474,432	8	200.00	2,838,528	26	250.00
1920×1080	[90]	1,695,744	—	100.00	1,216,512	11	250.00

Notably, the Symmetric LHT architecture consumes more DSP slices than several of the architectures. For example, the work presented in [20] and [21] use the Gradient LHT, which reduces the accuracy of line detection but consumes fewer DSP slices. In comparison, the Symmetric LHT applies votes for all angles in θ, which requires more DSP slices. The work in [89] and [13] uses fewer DSP slices than the Symmetric LHT because these architectures use the FIHT2 algorithm (or a variation of the algorithm). The FIHT2 algorithm does not require multiplications, but detection accuracy is reduced as erroneous results can occur after a few iterations.

The operational clock frequency for each architecture in Table 5.11 is also reported. For these particular implementations of the Symmetric LHT, the target clock frequency was set to 250 MHz. This operational frequency is an improvement in comparison to all other related works, except for the LHT implementation in [20].

5.4.5 Discussion of Results

Overall, the Symmetric LHT has reduced the memory requirements of the HPS compared to the parallel LHT without modifying the underlying algorithm. The memory bit-packing scheme employed by the Symmetric LHT effectively reduces BRAM tile requirements for most image resolutions. The Symmetric LHT also exhibits better memory consumption than the LHT architectures described in related works. It would be interesting to explore the effect of combining the Symmetric LHT with the FIHT2 architecture or the technique described in [21] for efficiently drawing lines.

To exploit the bit-packing capabilities of the Symmetric LHT on an FPGA, there must be 36 Kb BRAM primitives or a similar memory component available on the target device. This type of memory is useful as it allows the designer to trade-off the size of the BRAM's wordlength with the number of addressable locations, which is required to optimise the memory consumption of the bit-packed accumulator. The Symmetric LHT is not only applicable to AMD FPGAs. The symmetric Hough kernel and bit-packed accumulator can also be implemented on Intel FPGAs, which contain memory tiles that can be reshaped to improve memory allocation [116]. Note that the Symmetric LHT has only been tested using AMD FPGAs in this work.

It is useful to compare the Symmetric LHT with the standard LHT to evaluate the memory allocation of the HPS and determine if further optimisations should be investigated. Table 5.12 presents this comparison, where the last column shows the total memory allocation of the Symmetric LHT used to store the HPS.

Table 5.12: The memory consumption of the standard LHT and Symmetric LHT.

Resolution (Pixels)	**Standard LHT**	**Symmetric LHT**			**Memory Over Allocated(%)**
	Total Bits	**BRAM**		**Total Bits**	
		18 Kb	**36 Kb**		
320 × 240	648,000	91	—	1,677,312	258.84
333 × 333	764,640	91	—	1,677,312	219.36
512 × 512	1,306,800	2	89	3,317,760	253.88
800 × 600	1,800,000	2	89	3,317,760	184.32
1024 × 768	2,534,400	89	91	4,995,072	197.09
1280 × 720	2,910,600	89	91	4,995,072	171.62
1920 × 1080	4,760,640	180	91	6,672,384	140.16

For an image of 1920 × 1080 pixels, the Symmetric LHT inefficiently allocates 140.16% of the necessary memory resources to storing the HPS. This overallocation is significant as storing the HPS requires 58.01% of all BRAM tiles on the XCZU7EV-2E device. The memory allocation for the HPS in FPGA devices can still be improved to enable the implementation of the LHT on memory constrained FPGA systems.

The memory consumption for the parallel LHT, Symmetric LHT, and standard LHT can be presented graphically using a bar chart, as in Figure 5.18. Notably, the memory consumption of the Symmetric LHT is better than or equivalent to the parallel LHT across several image resolutions. However, the Symmetric LHT still requires significant memory resources in comparison to the standard LHT. Memory consumption could be improved by modifying the LHT algorithm to reduce the memory allocation of the HPS. Chapter 6 of this thesis presents a lossy compression scheme for reducing the memory requirements of the HPS and improving overall memory consumption in comparison to the standard LHT.

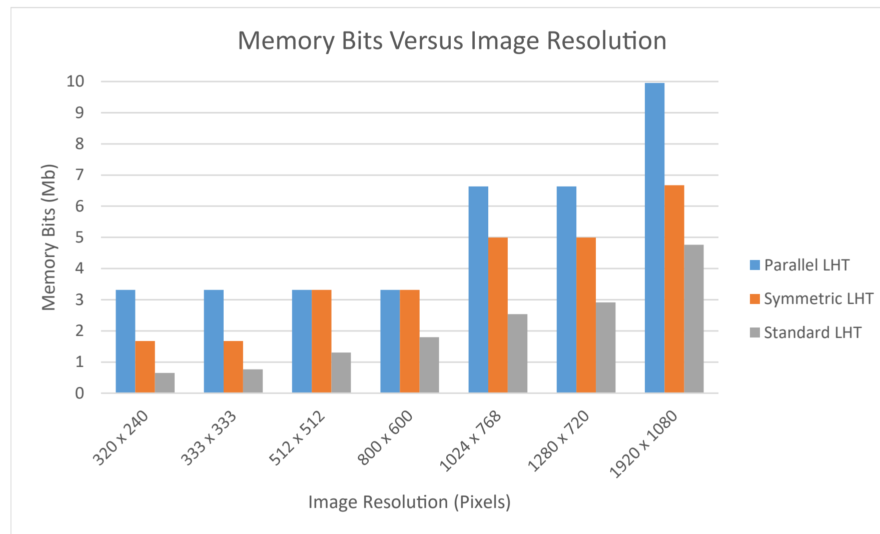


Figure 5.18: A bar chart comparing the number of memory bits required by each LHT algorithm across several image resolutions.

5.5 Conclusion

This chapter has detailed the Symmetric LHT, a memory-efficient FPGA architecture of the LHT that exploits spatial domain symmetry and bit-packing schemes. At the beginning of this chapter, an overview of the Symmetric LHT was presented. A new symmetrical coordinate system that reduces the computation of the Hough parameters was introduced. The memory-efficient, bit-packed accumulator was described using BRAM allocation tables. The Symmetric LHT architecture design was presented, which included detailed diagrams and descriptions of the pixel packing system, coordinate calculator, phase controller, symmetric Hough kernel, and bit-packed accumulator. Finally, the Symmetric LHT was evaluated for its FPGA resource consumption and operational clock frequency for several image resolutions. For an image of 1920×1080 pixels, the Symmetric LHT required 6,672,384 bits of memory (181 BRAMs). The Symmetric LHT saved 32.96% of memory in comparison to the parallel LHT architecture, which required 9,953,280 bits of memory (270 BRAMs). Additionally, the Symmetric LHT was found to be more memory-efficient than other FPGA architectures of the LHT reported in previously published work.

The Symmetric LHT can save memory resources and reduce computational requirements compared to the parallel LHT without affecting the accuracy of line detection. However, the underlying algorithm will need to be modified to further improve the memory consumption of the LHT in FPGA devices. There are four primary findings and outcomes from the work undertaken in this chapter. These are listed as follows:

1. This chapter presented the Symmetric LHT, which is unique in its approach to applying the LHT compared to previously published works, as it is the first to exploit spatial domain symmetry and bit-packing techniques to reduce the computational complexity and memory requirements of line detection.

2. Compared with the parallel LHT and other FPGA architectures of the LHT in previously published literature, the Symmetric LHT has similar or fewer memory requirements for storing the HPS. For an image of 1920×1080 pixels, the Symmetric LHT saves 32.96% of memory resources compared to the parallel LHT.

3. For an image of 1920×1080 pixels, the Symmetric LHT inefficiently over allocates 140.16% of the necessary memory resources to storing the HPS. This overallocation can be improved by modifying the LHT algorithm to reduce the memory allocation of the HPS. Chapter 6 presents a lossy compression technique to optimise further the memory required to implement the LHT on FPGA devices.

4. The Symmetric LHT is suitable for small FPGA devices that contain 36 Kb BRAM tiles and DSP slices. For example, a small FPGA containing these specialised resources can apply the Symmetric LHT (where $\delta_\theta = 1^\circ$ and $N_\theta = 180$) to an edge image of 1920×1080 pixels. However, the FPGA must contain 181 BRAM tiles and 45 DSP slices. In contrast, the parallel LHT requires 270 BRAM tiles and 89 DSP slices for the same image resolution, which typically requires a mid-range or large FPGA containing hundreds of thousands of logic elements. Embedded applications that use small FPGAs can take advantage of cheaper device costs, smaller packages, and lower energy consumption.

Chapter 6

The Angular-Regions Line Hough Transform

6.1 Introduction

A challenging problem with implementing the LHT on FPGAs is the inefficient allocation of memory resources to store the HPS. Many FPGA devices are incapable of implementing the LHT due to their limited on-chip memory. Few studies have investigated the memory requirements of the HPS. Previous work presented in [23] follows a divide-and-conquer approach by separating the candidate image into equal-sized sub-images before applying the LHT. This operation reduces the size of the HPS across the ρ-axis, which decreases memory requirements. This work can potentially be improved in terms of memory efficiency by reducing the HPS along the θ-axis instead.

This chapter presents a novel algorithmic approach that compresses the HPS to reduce its memory requirements in FPGA devices. This algorithm is named the Angular-Regions Line Hough Transform (ARLHT). The ARLHT decomposes the spatial image domain into regions of line orientations, affecting the dimensionality of the HPS by reducing its size across the θ-axis. The FPGA architecture of the ARLHT will be described in this chapter, and an evaluation of its resource consumption and processing time results will be given. Finally, this chapter compares the ARLHT architecture with previously published work and details the limitations of the proposed algorithm.

6.2 The ARLHT Algorithm

In this section, the ARLHT algorithm is detailed and its memory consumption in AMD FPGA devices is calculated. A functional block diagram illustrating the ARLHT algorithm is presented in Figure 6.1.

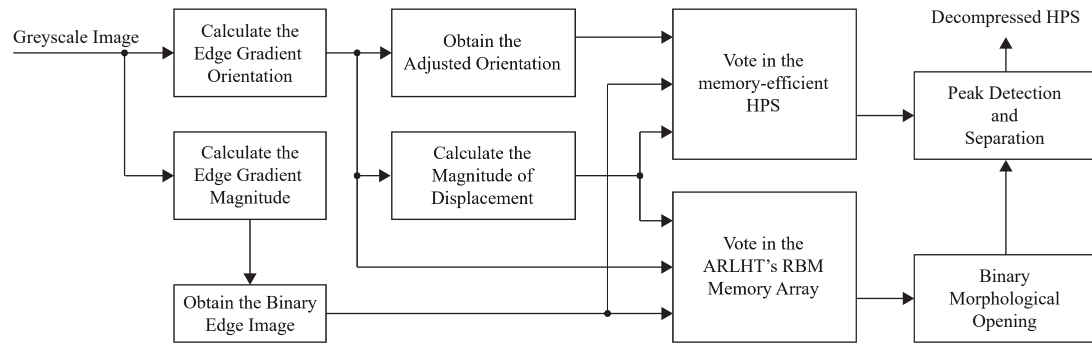
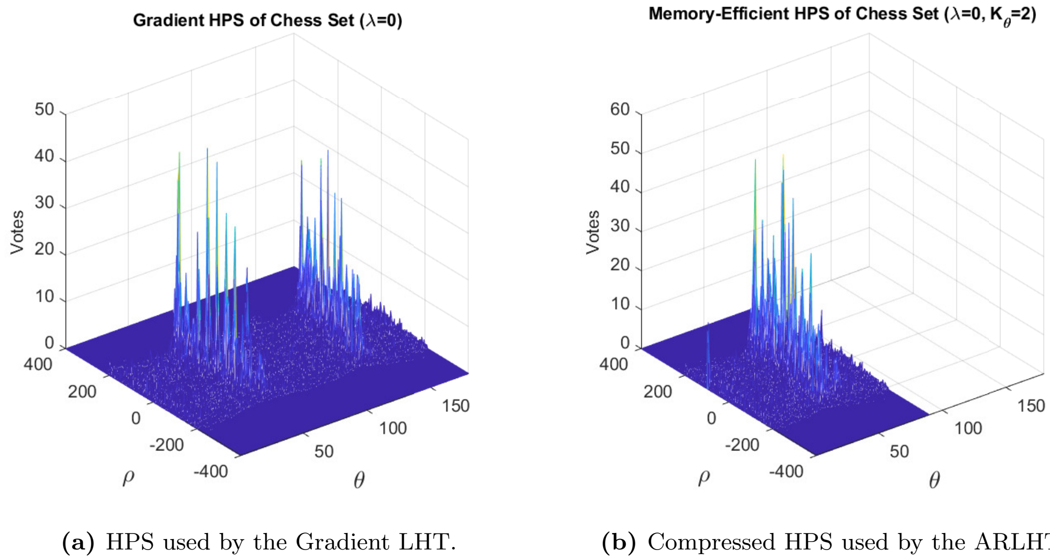


Figure 6.1: Functional block diagram of the ARLHT algorithm.

The gradient orientation of the input edge image, α, is used by the ARLHT to reduce the number of votes that are applied to the HPS. In this voting scheme, only one vote is applied to the HPS per edge pixel. Upon calculating the gradient orientation of an edge pixel, an adjusted orientation denoted as β is also derived. The magnitude of displacement and the adjusted orientation are combined to form the Hough parameters (ρ, β). These are used to apply votes in the HPS, where the θ-axis has been reduced in size. Another memory array known as an RBM (Region Bitmap) is initialised to maintain a record of the magnitude of (ρ, α) for each vote that is applied to the HPS. The RBM is used later with a morphological opening to separate overlapping and merged peaks in the HPS. The ARLHT follows the general method outlined in [23] to compress the HPS. The remainder of this section will describe the ARLHT's memory-compressed HPS, voting scheme, peak separation algorithm, and line reconstruction in more detail.

6.2.1 The Memory-Compressed HPS

The HPS produced using the Gradient LHT generally consists of sparse peaks that are distributed throughout the memory array. A motivating reason for compressing the HPS is to decrease the sparsity of the array and reduce memory consumption. The memory-compressed HPS that is used by the ARLHT algorithm is reduced in size across the θ-axis. When comparing the memory-compressed HPS to that used by the Gradient LHT, the size of the θ-axis is reduced by an integer factor, K_θ. The difference between the HPS used by the Gradient LHT and the ARLHT can be visually inspected in Figure 6.2. The HPS arrays were generated using the chessboard image presented previously in Figure 3.7a.



(a) HPS used by the Gradient LHT. **(b)** Compressed HPS used by the ARLHT.

Figure 6.2: The HPS used by the Gradient LHT (a) contains peaks that are sparser than the HPS used by the ARLHT (b).

The HPS used by the Gradient LHT consists of $N_\rho \times N_\theta$ addressable locations. The memory-compressed HPS, used by the ARLHT, contains $N_\rho \times N_\theta / K_\theta$ addressable locations. It is only possible to apply votes to the memory-compressed HPS if their value of orientation falls into the range $[0, \gamma - \delta_\theta]$, where γ is the angular length given by $(\theta_{N_\theta - 1} + \delta_\theta)/K_\theta$. Inapplicable votes that do not fall into the above range are adjusted using their orientation value, so that they can be applied to the HPS correctly. This new orientation value is referred to as the adjusted orientation, β.

6.2.2 The Voting Scheme

The ARLHT algorithm follows a unique voting procedure. Initially, the memory-efficient HPS is defined by using a value of K_θ to reduce the size of the θ-axis. Edge pixels can apply votes to the HPS using their corresponding Hough parameters (ρ, β). To obtain the adjusted orientation, β, the inapplicable gradient orientation value, α, must be moved into the range $[0, \gamma - \delta_\theta]$. The adjusted orientation can be calculated using,

$$\beta = \alpha - \left\lfloor \frac{\alpha}{\gamma} \right\rfloor \gamma. \tag{6.1}$$

The memory-efficient HPS does not provide a unique location for every possible line in the candidate image. Voting edge pixels from different angular regions of the candidate image need to be identified and separated in the HPS. The RBM is a two dimensional array of size $N_\rho \times N_\theta \times 1$ bit, which is initialised to zero before voting begins. It is responsible for recording the magnitude of displacement and gradient orientation (ρ, α) of votes that have been applied to the memory-efficient HPS. Votes are recorded by changing the corresponding location in the RBM to a binary '1'. Figure 6.3 presents an example of an RBM that was produced by applying the ARLHT to the chessboard image (previously shown in Figure 3.7a).

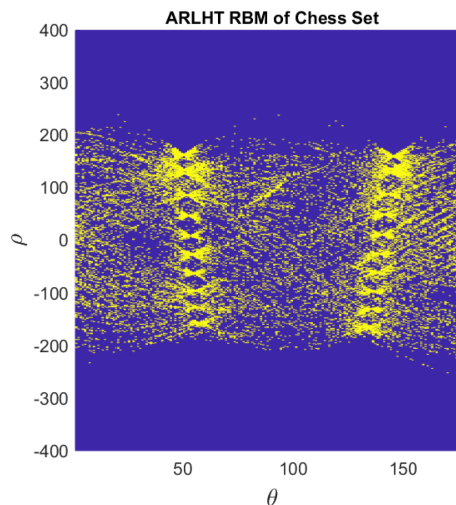


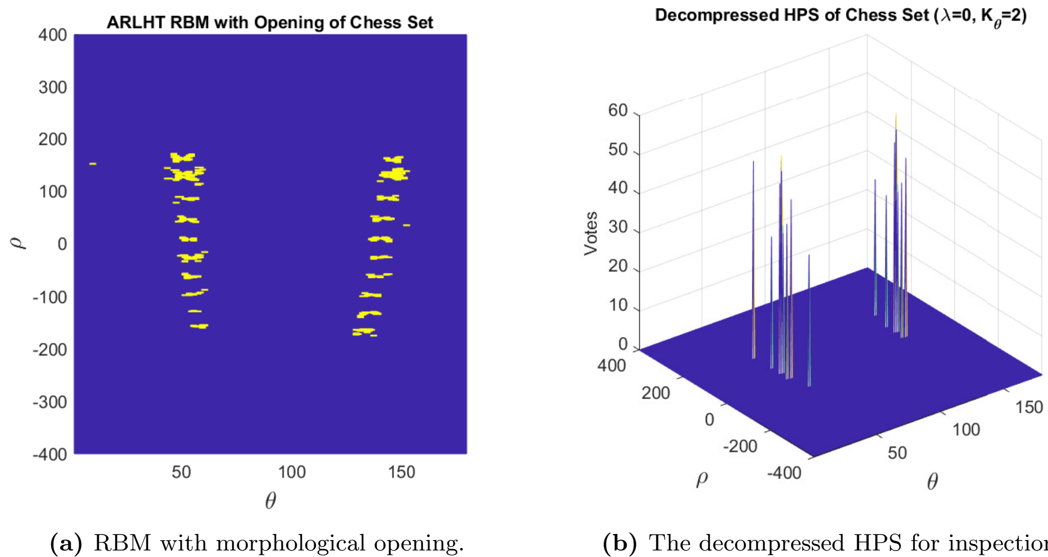
Figure 6.3: The RBM of the chessboard image using the ARLHT algorithm. Notice that only one memory array is required to store the RBM for the ARLHT.

6.2.3 Peak Separation and Detection

The memory-efficient HT presented in [23] uses an iterative peak separation and detection algorithm. Iterative algorithms are very suitable for general purpose processors as they can easily buffer image frames between algorithm iterations. In contrast, an iterative FPGA architecture requires a complicated memory design to achieve the same functionality. These architecture designs may also increase the output latency of the algorithm, which increases the overall processing time. Since the ARLHT will be implemented in an FPGA, another solution is required for peak separation and detection, rather than using an iterative approach.

The ARLHT uses the RBM and memory-efficient HPS to achieve peak separation and detection. The memory-efficient HPS is initially thresholded for peaks and the corresponding Hough parameters are extracted. The RBM is used to identify and cross-check the parameters of peaks that are extracted from the HPS, so that they can be assigned the correct value of orientation. The edge image may contain noise pixels whose Hough parameters are recorded on the RBM, which may produce spurious results during peak separation. To suppress this noise, the RBM is locally filtered using a morphological opening (previously described in Section 2.6.8). The result of applying an opening to the RBM for the chessboard image is given in Figure 6.4a. Notice that the noise has been suppressed in comparison to the original RBM shown in Figure 6.3. This method is a simple alternative to performing the peak separation algorithm proposed in [23] and is suitable for FPGA implementation.

An example of the ARLHT's peak separation and detection algorithm will now be described. A peak is located in the memory-efficient HPS at the location (ρ_i, β_i). If the RBM contains a binary '1' at the same location, then the corresponding Hough parameters can be used to locate a line in the input edge image. However, the RBM must also be searched across intervals of γ for the same value of β_i, to determine whether lines in other angular regions exist. If a binary '1' exists in any of these locations, then the corresponding Hough parameters should be used to locate a line in the input edge image. This procedure can be used to decompress the memory-efficient HPS for inspection purposes. The decompressed HPS is presented in Figure 6.4b.



(a) RBM with morphological opening.

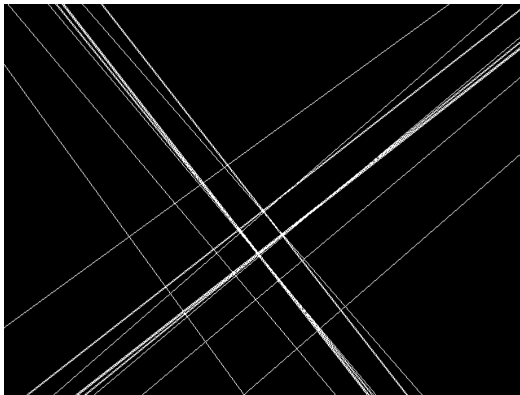
(b) The decompressed HPS for inspection.

Figure 6.4: The ARLHT RBM with morphological opening is presented in (a). The decompressed HPS of the chessboard is given in (b). The vote threshold used to generate the decompressed HPS is 60% of the maximum vote.

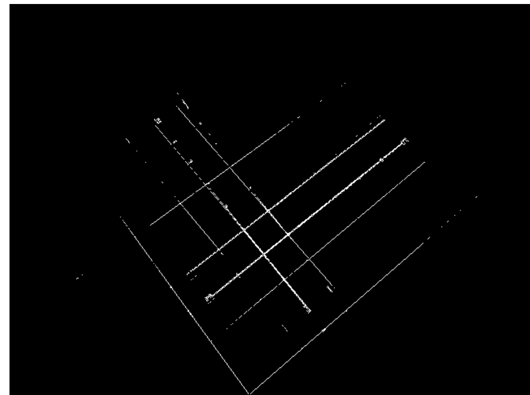
The peak separation and detection stage has identified Hough parameters that can be used during line reconstruction. While many of the identified parameters will correspond to lines in the original edge image, it is possible that spurious lines may form when the input edge image contains high levels of noise. The limitations of the ARLHT algorithm are discussed further in Section 6.4.5.

6.2.4 Line Reconstruction

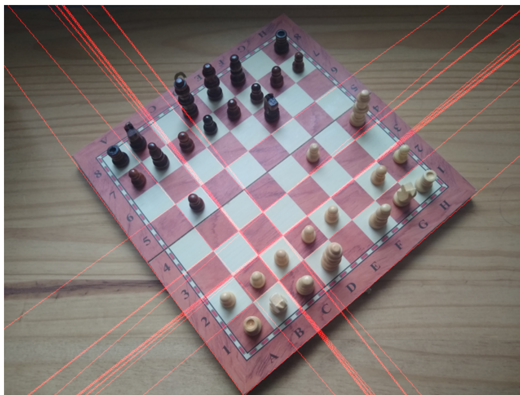
The Hough parameters that were extracted during the peak separation and detection stage now undergo line reconstruction using (3.6) and (3.7). The infinite and finite line images are presented in Figure 6.5a and Figure 6.5b, respectively. The infinite line image is overlaid onto the original colour image of the chessboard in Figure 6.5c and the greyscale image of the chessboard in Figure 6.5d. Notice that there are significantly fewer lines detected using the ARLHT in comparison to the standard LHT (given previously in Figure 3.10). Fewer lines are detected since the ARLHT uses the Gradient LHT ($\lambda = 0°$), which reduces the sensitivity of line extraction since fewer votes are applied to the HPS per edge pixel.



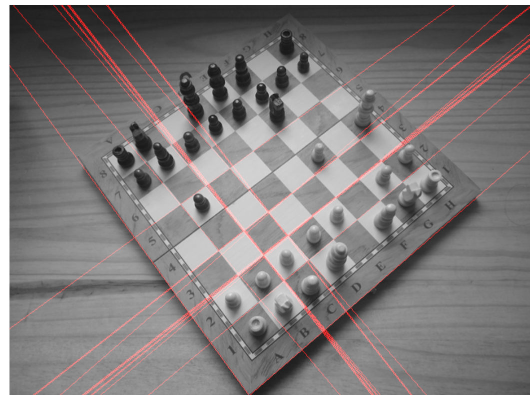
(a) Reconstructed line image of the chessboard created using (3.6) and (3.7).



(b) Reconstructed image combined with the original edge image using a logical AND operation.



(c) Overlay of the reconstructed image and the original colour image.



(d) Overlay of the reconstructed image and the original greyscale image.

Figure 6.5: Line reconstruction results for the ARLHT algorithm.

6.2.5 Memory Requirements

The total number of memory bits required by the memory-compressed HPS and RBM can be calculated for an $M \times N$ image using (6.2). Notice that each cell of the HPS must be represented by $\lceil \log_2(DK_\theta) \rceil$ bits. This is required as the number of votes that can be applied to a single location in the HPS has now increased by a factor of K_θ.

$$b = \frac{N_\rho N_\theta \lceil \log_2(DK_\theta) \rceil}{K_\theta} + N_\rho N_\theta \tag{6.2}$$

Later in Section 6.4.4, the memory requirements of the ARLHT are compared with existing algorithms and FPGA implementations.

6.3 The ARLHT Architecture Design

In this section, an FPGA architecture of the ARLHT algorithm is described. This architecture design will be used to evaluate the ARLHT's resource consumption and timing closure when implemented on the XCZ7UEV-2E device. In this FPGA implementation, $\delta_\rho = 1$, $\delta_\theta = 1°$, and the number of angular regions is set to $K_\theta = 4$. The operational range of θ is over $[0°, 179°]$. The ARLHT architecture was designed using the HEP workflow and reference design, which was described previously in Chapter 4. Before the system integration stage occurs, the ARLHT architecture can be configured to almost any image resolution. The architecture described in this section is configured to process an image of 1920×1080 pixels.

The remainder of this section will describe the architecture of the ARLHT, which includes its Hough kernel, adjusted orientation processor, accumulator circuit design, RBM circuit design, and peak separation and detection stage. A system overview of the ARLHT architecture design is shown in Figure 6.6.

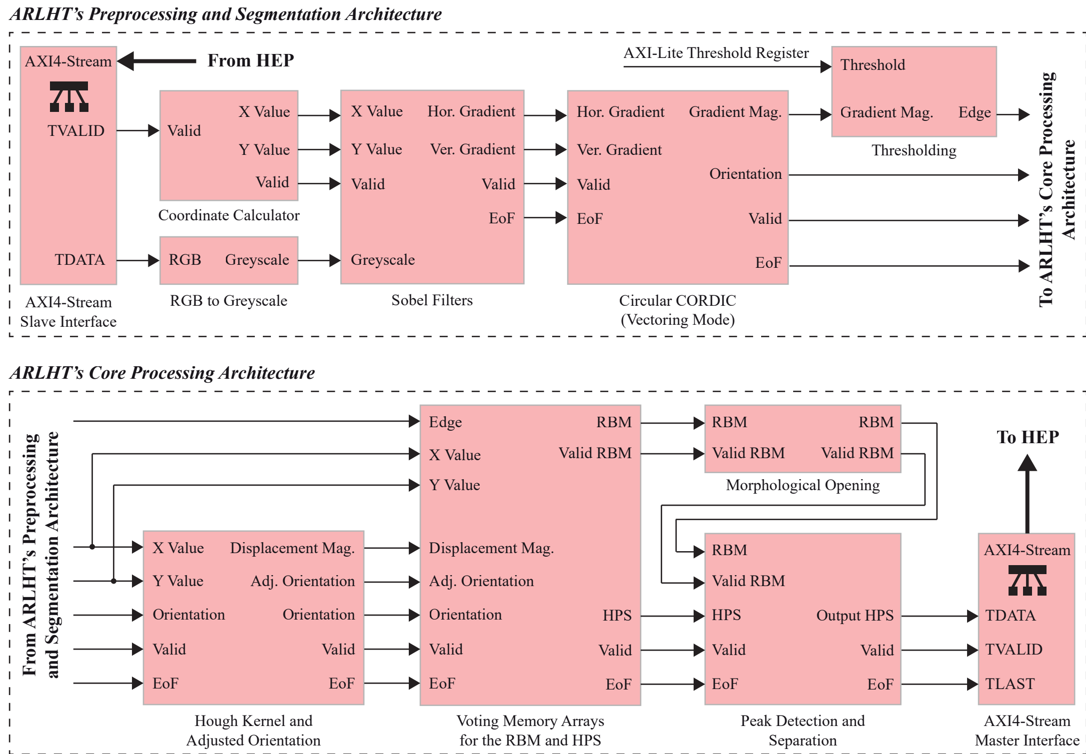


Figure 6.6: System overview of the ARLHT architecture design.

6.3.1 Preprocessing and Segmentation

The upper part of Figure 6.6 presents the ARLHT's preprocessing and segmentation architecture, which interfaces to the HEP's read DMA. The candidate test image is transferred into the ARLHT using the RGB colour format that is initially converted to greyscale. The coordinate calculator maintains track of the image position and the horizontal and vertical Sobel filters are applied to the greyscale image to obtain the horizontal and vertical gradient images. The separable filter architecture presented in Figure 2.24 is used to implement Sobel edge detection.

The gradient images are sent to a Circular CORDIC architecture that operates in vectoring mode. This instance of CORDIC calculates the gradient magnitude and orientation images similar to that presented in Section 2.6.7. Finally, the gradient magnitude is segmented using a threshold operation to produce binary edge pixels. The edge image and gradient orientation image are sent to the ARLHT's core processing architecture to obtain the memory-compressed HPS.

6.3.2 Hough Kernel and Adjusted Orientation

A Hough kernel architecture is required to compute the magnitude of displacement, ρ, using the gradient orientation, α. It is not necessary to design a parallel Hough kernel such as that given in Appendix B as only one set of Hough parameters is required per edge pixel. The Hough kernel for the ARLHT is presented in Figure 6.7. Notice that two LUTs are used to store the values of $\cos(\theta)$ and $\sin(\theta)$, reducing the overall computation during runtime.

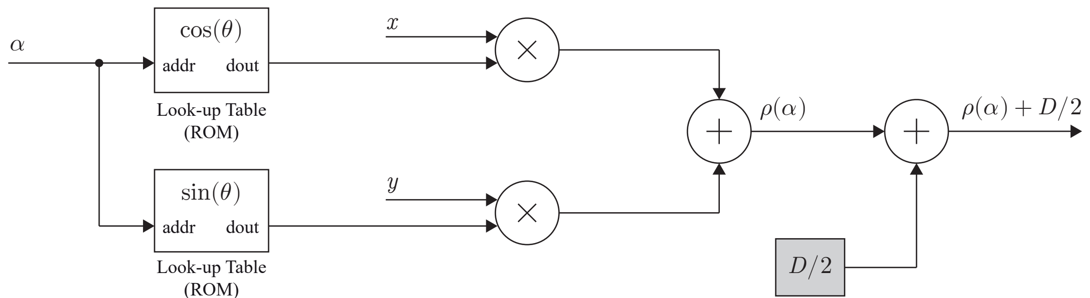


Figure 6.7: FPGA architecture of the ARLHT's Hough kernel. The output $\rho(\alpha)$ value is incremented by $D/2$ for memory addressing purposes.

After obtaining the Hough parameters (ρ, α) it is necessary to calculate the adjusted orientation, β. As mentioned previously, the adjusted orientation is used to apply votes in the memory-efficient HPS. It is adjusted such that β is in the range $[0, \gamma - \delta_\theta]$. The equation given in (6.1) can be used to calculate the adjusted orientation. However, this equation requires a division operation, which is not particularly suited to FPGAs. Division architectures consume significant resources and can limit timing performance as they cannot be efficiently pipelined. For this reason, a new architecture is presented in Figure 6.8, which is able to efficiently calculate the adjusted orientation.

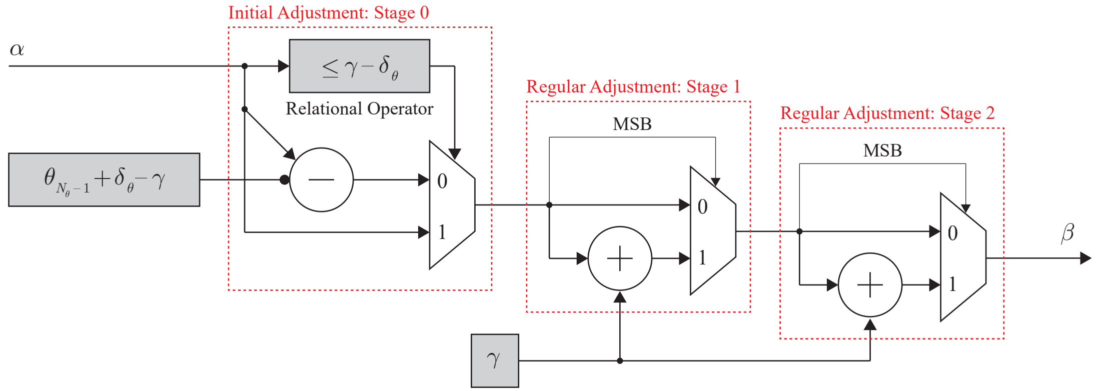


Figure 6.8: FPGA architecture for calculating the adjusted orientation when $K_\theta = 4$. As highlighted in the red box, there are $K_\theta - 1$ adjustment stages in total (one initial adjustment stage, and two regular adjustment stages).

The architecture in Figure 6.8 is able to calculate the adjusted orientation when $K_\theta = 4$. This architecture design comprises of an initial adjustment stage and two regular adjustment stages. The initial adjustment stage uses a relational operator to determine if the gradient orientation, α, is within the range $[0, \gamma - \delta_\theta]$. When true, the value of α is passed directly into the regular adjustment stage. Alternatively, $\theta_{N_\theta - 1} + \delta_\theta - \gamma$ is subtracted from α when it is not within range and sent to the regular adjustment stage. The initial adjustment stage is required by all ARLHT architectures when $K_\theta \geq 2$.

The regular adjustment stage is only required by ARLHT architectures when $K_\theta \geq 3$. This stage increments the output of the previous stage by γ if its value is negative. The regular adjustment stages are cascaded such that there are $K_\theta - 2$ instances, which are used to iteratively adjust α into the range $[0, \gamma - \delta_\theta]$.

6.3.3 Accumulator Circuit Design

The ARLHT's accumulator is used to store the memory-efficient HPS. Figure 6.9 presents the top-level accumulation architecture. As shown, the accumulator is controlled by an FSM that has been designed as a Moore machine. There is also an accumulator memory subsystem for storing the HPS.

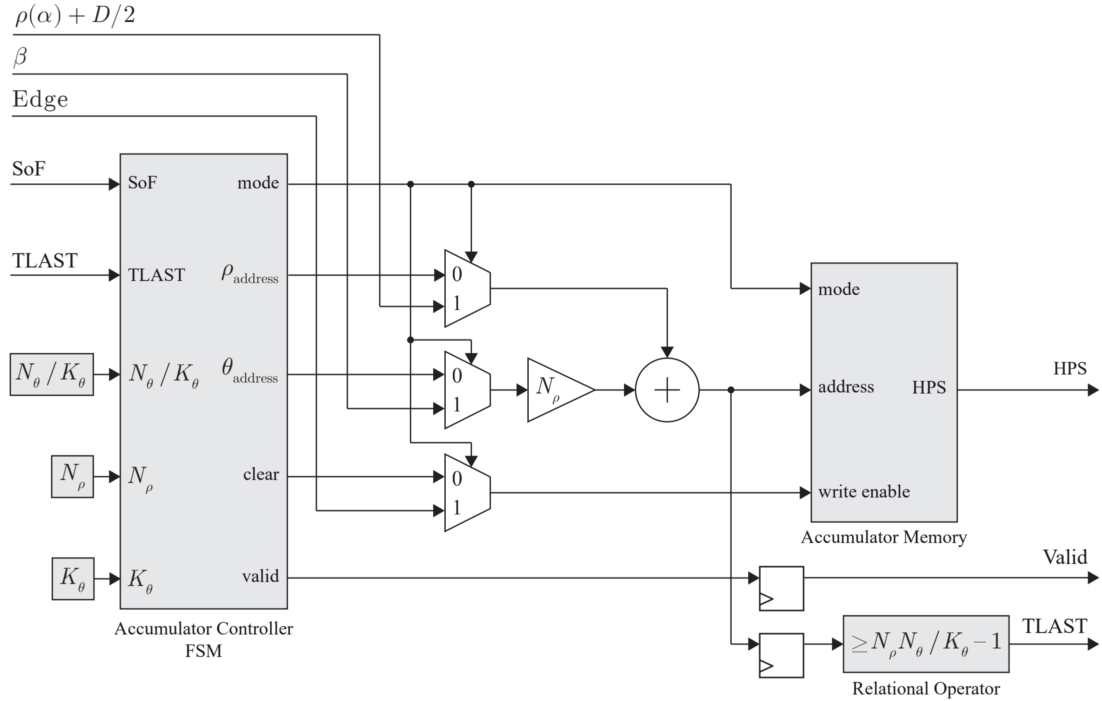


Figure 6.9: FPGA architecture for the ARLHT's top-level accumulator design. The accumulation controller implements an FSM to control the accumulator memory. Note that the TLAST output is only valid when the clear output of the accumulator controller is high.

The HPS is stored in the accumulator memory using a contiguous memory scheme. Accessing the array using the Hough parameters (ρ_i, θ_i) can be performed using $N_\rho \theta_i + \rho_i$ to generate an address. This functionality is implemented using a gain and adder in the middle of Figure 6.9. The accumulator memory subsystem is similar to that used by the Symmetric LHT to accumulate votes. However, the ARLHT only uses one instance of the accumulator memory. The architecture of the accumulator memory subsystem is presented in Figure 6.10 for inspection. Notice the accumulator feedback path that is highlighted by the dashed line.

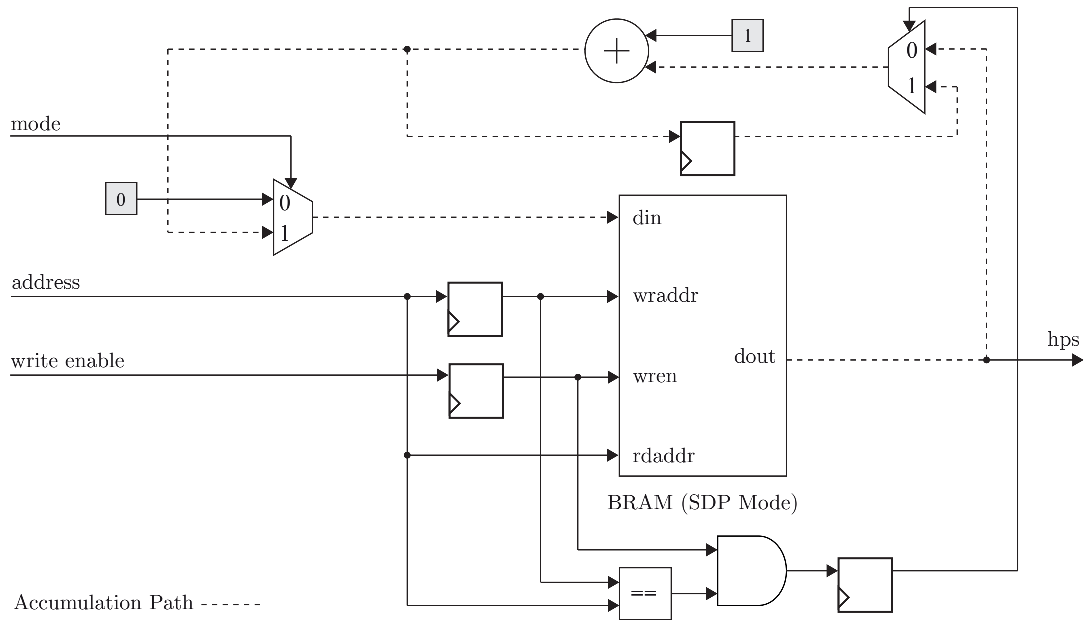


Figure 6.10: A circuit diagram of the ARLHT accumulator memory subsystem. The accumulation path is indicated by the dashed line.

The BRAM in the accumulator memory is configured to operate in SDP mode, which allows one read and write transaction to occur per clock cycle. To apply a vote, the data stored at a memory location is accessed using the 'rdaddr' port. After one clock cycle, the stored data is presented at the 'dout' port. The data is sent through the accumulation path where it is incremented by one. After voting, the data is sent to the BRAM's 'din' port. If a logical '1' is applied to the 'wren' port on the BRAM, then the data is written back into the original address location.

As described previously in Section 5.3.6, a read and write operation may occur at the same address, causing the BRAM to present old data at the 'dout' port rather than the new data to be written into memory. Votes can be unintentionally dropped if two or more consecutive edge pixels share the same address in memory. For this reason, the relational operator and the AND gate below the BRAM in Figure 6.10 check if the read and write addresses are the same. When they are both equal, the multiplexer on the right of Figure 6.10 loops data that was accessed in the previous clock cycle, back into the accumulation path. The accumulated vote is written back into memory as soon as the read and write addresses are no longer equal to one another.

The accumulator controller subsystem in Figure 6.9 contains the FSM to control the voting, reading, and clearing operations that are required by the accumulator memory. The input signals to the FSM are the SoF signal, the TLAST signal, and three constants, N_ρ, N_θ/K_θ, and K_θ. Additionally, there are four internal registers: the Current State register, a ρ counter register denoted as ρ_{count}, a θ counter register denoted as θ_{count}, and a K_θ counter register denoted as K_{count}. All counter registers are connected to the output of fixed point counters, which accumulate during the FSM's operation. The controller's states and transitions can be seen in the FSM diagram presented in Figure 6.11. Although not shown in the diagram, an undefined state will cause the FSM to transition back to the IDLE state.

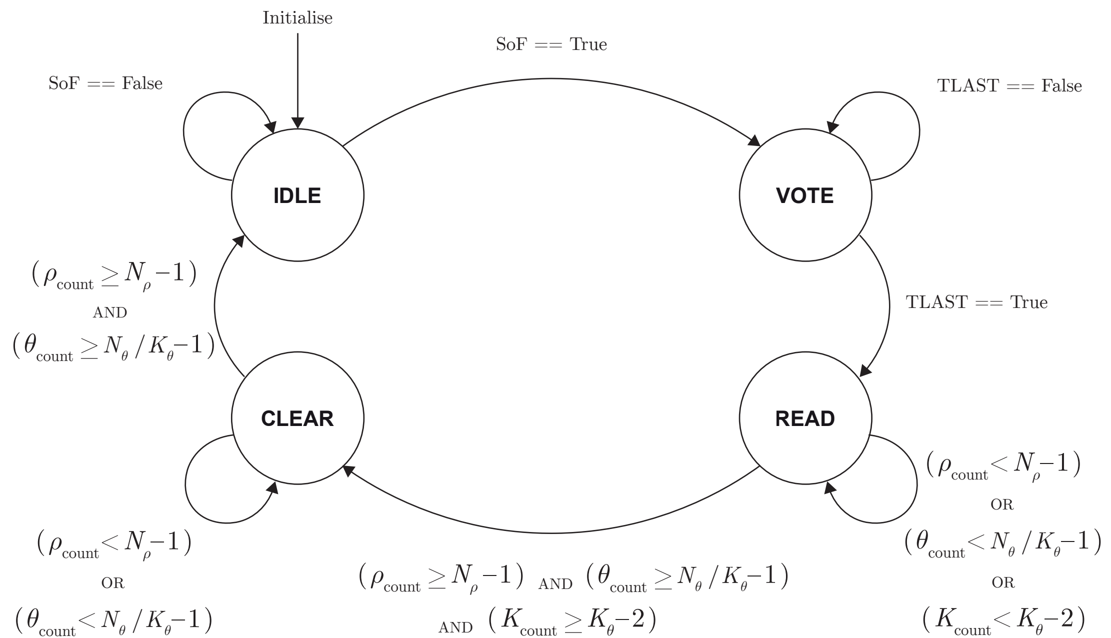


Figure 6.11: FSM diagram for the ARLHT's accumulator controller.

The FSM contains four states of operation: IDLE, VOTE, READ, and CLEAR. The reader is directed to Section 5.3.5 for a description of the IDLE, VOTE, and READ states. The CLEAR state in this particular FSM is unique, as it will be used to clear the accumulator memory for the next image and read out the existing HPS at the same time. The remainder of this section describes the functionality of the FSM and the design of the CLEAR state further.

In the IDLE state, the FSM waits for the SoF signal to rise and then transitions into the VOTE state. After voting, the FSM transitions into the READ state when the TLAST input is set high. While in the READ state, the FSM will generate addresses to read out the memory-efficient HPS from the accumulator memory. The FSM will perform the read operation $K_\theta - 1$ times before transitioning into the CLEAR state. While in the CLEAR state, the FSM reads out the memory-efficient HPS one last time, while clearing each memory location for the next image. Since the HPS is K_θ times smaller than the RBM, then it is necessary to read the HPS from memory K_θ times. The peak separation and detection stage will compare the HPS to the filtered RBM across intervals of size γ.

The FSM contains five output signals: mode, clear, valid, ρ_{address}, and θ_{address}. The outputs are configured as presented in Table 6.1 and depend on the given state. The mode, clear, and valid outputs are each set to either False or True, depending on the given state. The ρ_{address} and θ_{address} outputs are always set to the value of the ρ_{count} and θ_{count} registers, respectively. Notice that the valid signal is high in the clear state, which indicates that the HPS is being read from memory.

Table 6.1: Accumulator controller output values for the ARLHT.

Output	**IDLE**	**VOTE**	**READ**	**CLEAR**
mode	False	True	False	False
valid	False	False	True	True
clear	False	False	False	True
$\boldsymbol{\rho_{\text{address}}}$	ρ_{count}	ρ_{count}	ρ_{count}	ρ_{count}
$\boldsymbol{\theta_{\text{address}}}$	θ_{count}	θ_{count}	θ_{count}	θ_{count}

The internal registers of the FSM will now be described. The first of these is the Current State register, which holds the FSM state. The value of the Current State register and the conditions required to influence the next state can be seen in Table 6.2. As shown, when the Current State register is IDLE, then the next possible state is VOTE. The next state is selected based on the condition column i.e. SoF == True. In this example, if the SoF remains False, then the condition is not met and the Current State register remains in the IDLE state. Table 6.2 is a useful way of representing the FSM diagram presented in Figure 6.11.

Table 6.2: Accumulator controller Current State register for the ARLHT.

Register	Value	Next Value	Condition
Current State	IDLE	VOTE	SoF == True
	VOTE	READ	TLAST == True
	READ	CLEAR	$(\rho_{\text{count}} \geq N_\rho - 1)$ and $(\theta_{\text{count}} \geq N_\theta/K_\theta - 1)$ and $(K_{\text{count}} \geq K_\theta - 2)$
	CLEAR	IDLE	$(\rho_{\text{count}} \geq N_\rho - 1)$ and $(\theta_{\text{count}} \geq N_\theta/K_\theta - 1)$

The configuration of the remaining FSM registers can also be described in a similar way to the Current State register. The register values for a given state and the required conditions are presented in Table 6.3. Notice that the K_{count} register is not required to clear the accumulator memory.

Table 6.3: Accumulator controller ρ_{count}, θ_{count}, and K_{count} registers for the ARLHT.

Register	State	Value	Next Value	Condition
ρ_{count}	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	READ	0	1	$\rho_{\text{count}} < N_\rho - 1$
		1	2	$\rho_{\text{count}} < N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\rho - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
	CLEAR	0	1	$\rho_{\text{count}} < N_\rho - 1$
		1	2	$\rho_{\text{count}} < N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\rho - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
θ_{count}	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	READ	0	1	$\rho_{\text{count}} \geq N_\rho - 1$
		1	2	$\rho_{\text{count}} \geq N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\theta/K_\theta - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
	CLEAR	0	1	$\rho_{\text{count}} \geq N_\rho - 1$
		1	2	$\rho_{\text{count}} \geq N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\theta/K_\theta - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
K_{count}	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	READ	0	1	$(\rho_{\text{count}} \geq N_\rho - 1)$ and $(\theta_{\text{count}} \geq N_\theta/K_\theta - 1)$
		1	2	$(\rho_{\text{count}} \geq N_\rho - 1)$ and $(\theta_{\text{count}} \geq N_\theta/K_\theta - 1)$
		$\vdots$	$\vdots$	$\vdots$
		$K_\theta - 2$	0	$(\rho_{\text{count}} \geq N_\rho - 1)$ and $(\theta_{\text{count}} \geq N_\theta/K_\theta - 1)$
	CLEAR	0	0	No Condition

The ρ_{count} and θ_{count} registers operate in a similar manner to that described previously in Section 5.3.5 for the Symmetric LHT. The primary change made in the FSM used by the ARLHT is the addition of the K_{count} register. This register maintains a record of the number of times the memory-efficient HPS has been read from memory. In the read state, the K_{count} register is incremented when the ρ_{count} and θ_{count} registers are both at their maximum values.

6.3.4 RBM Circuit Design

Implementing the RBM on an FPGA is far simpler than the accumulator array. However, the RBM still requires many of the fundamental components that have been identified so far including an RBM controller (FSM), an RBM memory, and an address generator. An overview of the RBM circuit is presented in Figure 6.12.

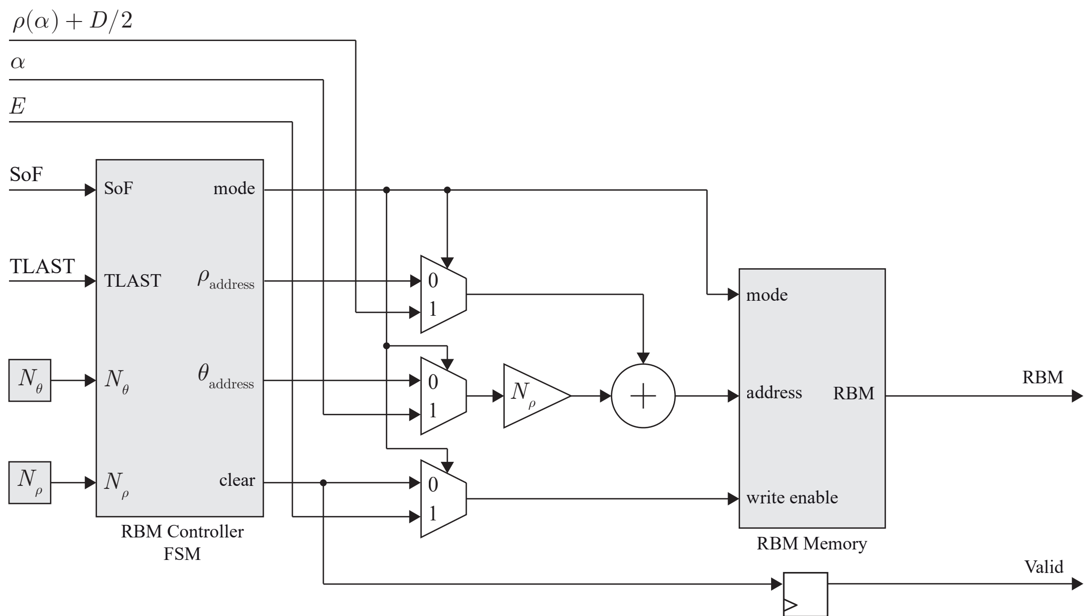


Figure 6.12: FPGA architecture for the ARLHT's top-level RBM circuit design. The RBM controller implements an FSM to control the RBM memory.

The RBM controller subsystem requires four input signals. These are the SoF signal, the TLAST signal, and two constants, N_ρ and N_θ. The FSM only requires three states to control the RBM memory. These states are IDLE, VOTE, and CLEAR. The IDLE state waits for a rising edge on the SoF input, which then transitions the FSM

into the VOTE state. After RBM voting, the FSM transitions into the CLEAR state. At this point, the FSM will read through the entire RBM memory and clear it for the next image. While the memory is cleared, the RBM is sent to the peak separation and detection stage. The FSM diagram is illustrated in Figure 6.13.

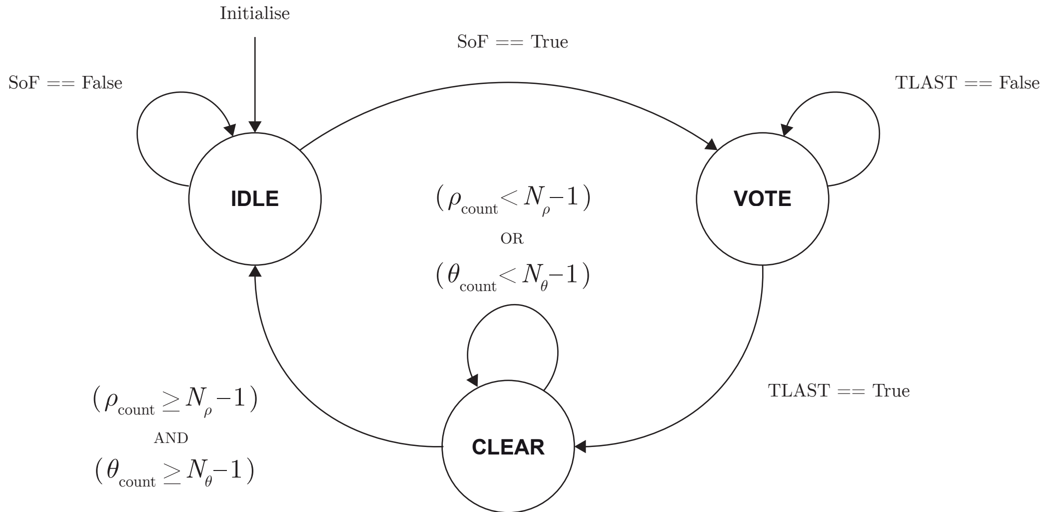


Figure 6.13: FSM diagram for the ARLHT's RBM controller.

There are three internal registers inside the FSM: the Current State register, a ρ counter register denoted as ρ_{count}, and a θ counter register denoted as θ_{count}. All counter registers are connected to the output of fixed point counters, which accumulate during the FSM's operation. The FSM contains four output signals that are named mode, clear, ρ_{address}, and θ_{address}. The mode output determines whether the RBM memory is configured to vote, or clear its memory contents. The clear output determines whether locations in the RBM memory are set to zero. The ρ_{address} and θ_{address} outputs are always set to the value of the ρ_{count} and θ_{count} registers, respectively. Table 6.4 presents the output values of the FSM for a given state.

Table 6.4: RBM controller output values for the ARLHT.

Output	**IDLE**	**VOTE**	**CLEAR**
mode	False	True	False
clear	False	False	True
$\rho_{\textbf{address}}$	ρ_{count}	ρ_{count}	ρ_{count}
$\theta_{\textbf{address}}$	θ_{count}	θ_{count}	θ_{count}

The Current State register, which holds the FSM state, will now be described. The value of the Current State register and the conditions required to influence the next state can be seen in Table 6.5. This table is synonymous to the FSM diagram presented in Figure 6.13. For example, when the Current State register is IDLE and the SoF input is high, then the next state is VOTE. However, if the SoF input remains low, then the Current State register remains in the IDLE state.

Table 6.5: RBM controller Current State register for the ARLHT.

Register	**Value**	**Next Value**	**Condition**
Current State	IDLE	VOTE	SoF == True
	VOTE	CLEAR	TLAST == True
	CLEAR	IDLE	$(\rho_{\text{count}} \geq N_\rho - 1)$ and $(\theta_{\text{count}} \geq N_\theta - 1)$

The remaining FSM registers that store the ρ_{count} and θ_{count} values can be described using a similar method in Table 6.6.

Table 6.6: RBM controller ρ_{count} and θ_{count} registers.

Register	**State**	**Value**	**Next Value**	**Condition**
$\boldsymbol{\rho}_{\textbf{count}}$	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	CLEAR	0	1	$\rho_{\text{count}} < N_\rho - 1$
		1	2	$\rho_{\text{count}} < N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\rho - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$
$\boldsymbol{\theta}_{\textbf{count}}$	IDLE	0	0	No Condition
	VOTE	0	0	No Condition
	CLEAR	0	1	$\rho_{\text{count}} \geq N_\rho - 1$
		1	2	$\rho_{\text{count}} \geq N_\rho - 1$
		$\vdots$	$\vdots$	$\vdots$
		$N_\theta - 1$	0	$\rho_{\text{count}} \geq N_\rho - 1$

The ρ_{count} and θ_{count} registers operate in a similar way to the internal counter registers of the accumulator's FSM. In the IDLE and VOTE states, the ρ_{count} and θ_{count} registers are set to zero. In the CLEAR state, the ρ_{count} is incremented by one until it is equal to $N_\rho - 1$. When this condition occurs, the ρ_{count} register is set to zero. Meanwhile, the θ_{count} register increments by one. This process continues until the θ_{count} register reaches $N_\theta - 1$. At this point, the ρ_{count} and θ_{count} registers are both set to zero and the Current State register transitions to IDLE as defined in Table 6.5.

Finally, the remaining section of the RBM circuit design to discuss is the RBM memory. This memory architecture is less complex than the accumulator memory subsystem. Figure 6.14 presents the architecture design for the RBM memory.

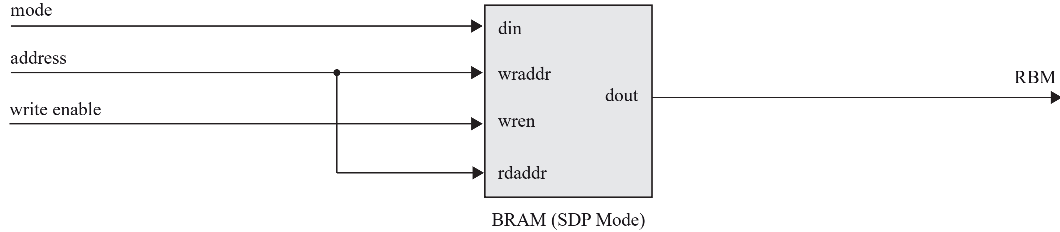


Figure 6.14: The RBM memory subsystem architecture that performs RBM voting. The BRAM selected is configured in SDP mode.

The mode input of the RBM memory is used to determine if a binary '0' or '1' will be written to a memory location that has been accessed using the address port. Since there is no accumulation of data occurring in the RBM memory subsystem, then it is not necessary for the design to contain any other control logic components.

6.3.5 Peak Separation and Detection Circuit Design

Figure 6.15 presents the FPGA architecture for the peak separation and detection stage. The binary morphological subsystems are each implemented using the design given previously in Figure 2.28 on page 48. An erosion followed by a dilation applies a morphological opening to the RBM. The HPS stream also requires row buffering to delay match the system, as the RBM filters add latency to the design. After the morphological opening, the RBM separates peaks in the HPS using a multiplexer.

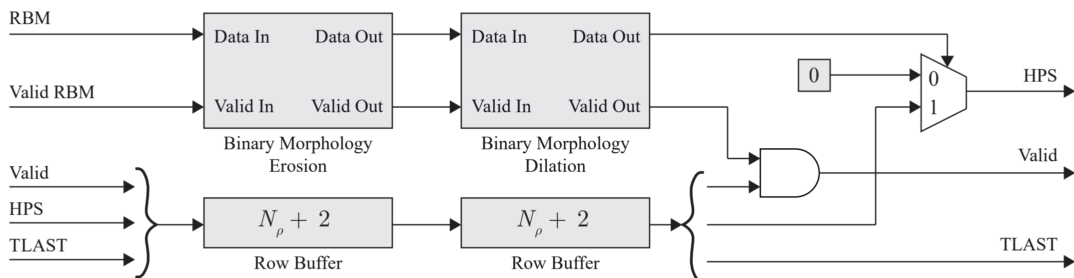


Figure 6.15: Peak separation and detection architecture for the ARLHT.

6.4 Analysis and Evaluation

This section presents the FPGA resource consumption and processing time results of the ARLHT architecture. The FPGA resource requirements are compared with the standard LHT and architectures from past works to highlight its resource efficiency. The implementation of the ARLHT using the HEP can be accessed electronically in [106].

6.4.1 Implementation Results

The ARLHT's core processing architecture that is presented in Figure 6.6 was implemented alongside the HEP. Synthesis and implementation of the ARLHT architecture was carried out using the Vivado Design Suite. The maximum clock frequency of the ARLHT architecture was reported to be 200 MHz. The ARLHT design was configured to process an image of 1920×1080 pixels and $\delta_\theta = 1°$ and $\delta_\rho = 1$. The FPGA resource consumption of the ARLHT architecture (without the HEP), was reported for the XCZU7EV-2E device, as shown in Table 6.7.

Table 6.7: FPGA resource consumption for the ARLHT architecture, which is implemented on the XCZU7EV-2E device.

Resource	Available	Used	Percentage (%)
LUTs	230,400	4,448	1.93
LUT RAM	101,760	3,027	2.97
FFs	460,800	4,503	0.98
BRAM	312	61	19.55
DSP48E2	1,728	0	0.00

The ARLHT does not use any DSP48E2 slices, as all multiplications are implemented using the FPGA logic fabric. The memory-efficient accumulator requires 48 36 Kb BRAMs and one 18 Kb BRAM to store the compressed HPS, while the RBM circuit consumes 12 36 Kb BRAMs and one 18 Kb BRAM. In total, the ARLHT requires 2,248,704 bits to process a 1920×1080 pixel image, while the standard LHT requires 4,760,640 bits to store the HPS in memory for the same image resolution. Overall, the ARLHT saves 52.76% of memory compared to the standard LHT. Note that the ARLHT only accumulates one vote per edge pixel, reducing the accuracy of line detection in comparison to the standard LHT (see Section 3.3.1).

The memory requirements of the ARLHT architecture can be directly compared with the standard LHT across a range of image resolutions. The ARLHT architectures were designed using the HEP, and the HPS was configured for $\delta_\rho = 1$, $\delta_\theta = 1°$, and $N_\theta = 180$. Table 6.8 presents the memory requirements for each architecture.

Table 6.8: The memory consumption of the standard LHT and the ARLHT architecture.

Resolution (Pixels)	Standard LHT	ARLHT Architecture			Memory Saved(%)
	Total Bits	BRAM		Total Bits	
		18 Kb	36 Kb		
320 × 240	648,000	1	11	423,936	34.58
333 × 333	764,640	1	13	497,664	34.92
512 × 512	1,306,800	—	20	737,280	43.58
800 × 600	1,800,000	1	27	1,013,760	43.68
1024 × 768	2,534,400	2	35	1,327,104	47.64
1280 × 720	2,910,600	2	40	1,511,424	48.07
1920 × 1080	4,760,640	2	60	2,248,704	52.76

The memory saved by the ARLHT in comparison to the standard LHT is given in the final column of Table 6.8. For all image resolutions, the ARLHT architecture efficiently allocates less memory than that required by the standard LHT. However, the memory efficiency of the HPS has limitations that must be addressed. These limitations are discussed further in Section 6.4.5.

6.4.2 Processing Time Results

The ARLHT architecture can be evaluated for its processing time when operating on images of different resolutions. The processing time of the ARLHT architecture for an image of 1920 × 1080 pixels is reported by the HEP as 12.37 ms (rounded to two decimal places). The corresponding frame rate of the ARLHT architecture is approximately 80.84 fps. This architecture design is able to process the FHD video standard.

Table 6.9 presents the processing time results of ARLHT architectures that are configured to process various image resolutions. The maximum clock frequency of each architecture is shown in column two of the table. The fastest frame processing time of 0.60 ms is achieved using the smallest resolution of 320 × 240 pixels.

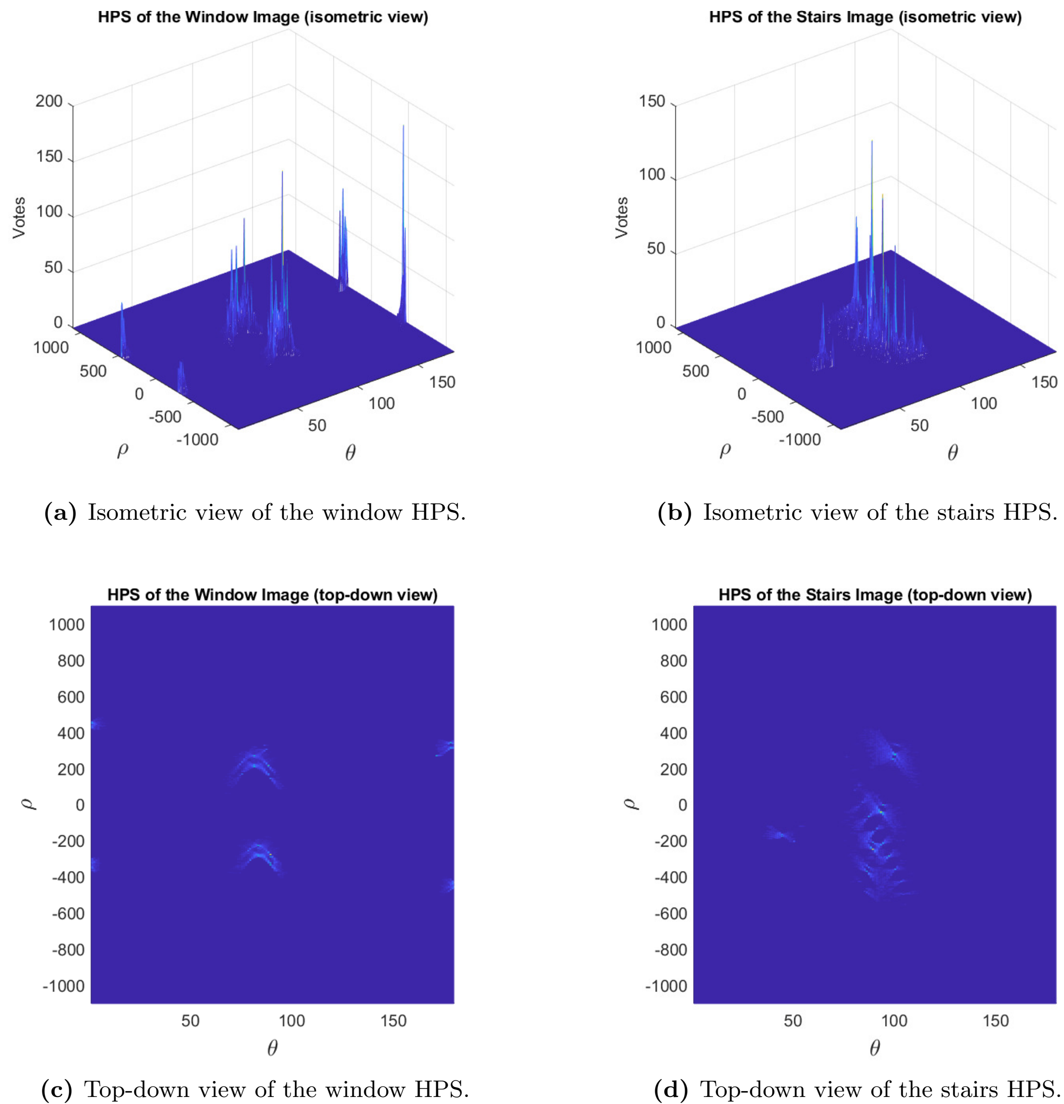
Table 6.9: Processing time results of ARLHT architectures that target various image resolutions. The processing time column contains measurement results from the HEP.

Resolution (Pixels)	Clock Frequency (MHz)	Processing Time (ms)	Frames Per Second (fps)
320×240	250.00	0.60	1666.66
333×333	250.00	0.79	1265.82
512×512	250.00	1.58	632.91
800×600	230.00	2.88	347.22
1024×768	225.00	4.53	220.75
1280×720	225.00	5.29	189.04
1920×1080	200.00	12.37	80.84

Notably, the maximum achievable clock frequency of the ARLHT architecture for image resolutions equal to or larger than 800×600 pixels is lower than the Symmetric LHT shown previously in Table 5.10. One possible cause of this decrease in clock frequency could be attributed to the spread of BRAM tiles that are distributed throughout a large area of the FPGA logic fabric. All BRAM tiles for the ARLHT are driven by the same address and data signals, unlike the Symmetric LHT. Spreading FPGA resources over large areas can increase routing delays between components, which leads to longer critical paths and lower operational clock frequencies.

6.4.3 Architecture Validation and Testing

Previously, Section 4.4.3 demonstrated the hardware validation procedure for FPGA architectures of the LHT using the HEP. The ARLHT architecture design described in this chapter was successfully hardware validated using the edge images of the window and stairs, given in Figure 4.14 on page 120. The HPS returned by the simulation and software model of the ARLHT were the same as the HPS output from the ARLHT architecture running on the target XCZU7EV-2E device. The HPS for each test image is given in Figure 6.16. Note that the total number of votes applied to the HPS using the ARLHT architecture will be significantly reduced in comparison to the standard LHT. Votes are reduced because each edge pixel votes once in the HPS.



(a) Isometric view of the window HPS.

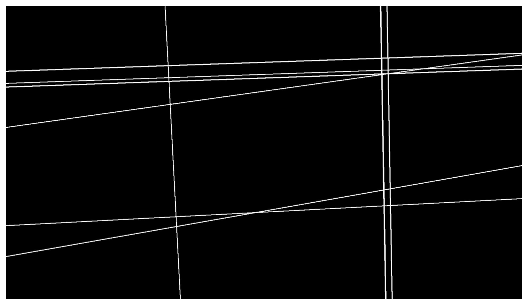
(b) Isometric view of the stairs HPS.

(c) Top-down view of the window HPS.

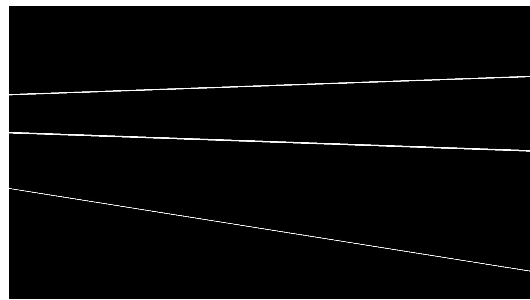
(d) Top-down view of the stairs HPS.

Figure 6.16: HPS results for the hardware validation of the ARLHT architecture on the XCZU7EV-2E device. The isometric view of the HPS for the window image (a), and the stairs image (b). The top-down view of the HPS for the window image (c), and the stairs image (d).

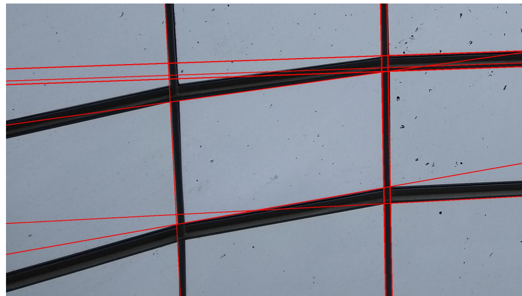
The parameters of peaks detected in the HPS for each test image are used in (3.6) and (3.7) to reconstruct lines. The ARLHT applies fewer votes to the HPS in comparison to the standard LHT. Therefore, the number of detected lines will be lower (see Section 3.3.1 for more information). The reconstructed line results are presented in Figure 6.17 for inspection.



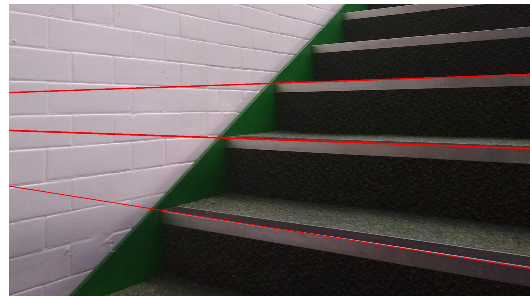
(a) Reconstructed line image of the input window edge image.



(b) Reconstructed line image of the input stairs edge image.



(c) Overlay of the reconstructed image and the original colour image of the window.



(d) Overlay of the reconstructed image and the original colour image of the stairs.

Figure 6.17: Line reconstruction results for the test images input into the ARLHT architecture. The reconstructed line images of the window (a) and stairs (b). The reconstructed lines overlaid on top of the original colour images of the window (c) and stairs (d).

The line reconstruction results demonstrate that the ARLHT has sufficient line detection accuracy, while consuming significantly less memory than the parallel LHT, Symmetric LHT, and standard LHT. The ARLHT was validated on physical target hardware using the Jupyter Lab environment. The validation notebooks can be inspected in Appendix D.

6.4.4 Comparison with Related Works

This section compares the memory efficiency of the ARLHT architecture with previously published implementations of the LHT. The ARLHT architectures will be configured to have the same, or similar, parameters and image resolution as other LHT architecture designs. For example, suppose a previously published LHT architecture can process an image resolution of 512×512 pixels and uses $\delta_\theta = 2°$. The ARLHT will also be configured to use these parameters to compare memory consumption directly.

The implementation of the custom ARLHT architectures described in this section can be obtained in [106].

The previously published LHT architectures have already been discussed in Section 6.4.4. Therefore, it is unnecessary to explore each LHT architecture again in detail. Instead, Table 6.10 compares the performance of each LHT architecture with an equivalent ARLHT architecture. The table presents the DSP slice requirements and memory consumption in terms of bits. The maximum achievable clock frequency of each architecture is also presented.

Table 6.10: Results of the ARLHT and comparison with related works.

Resolution (Pixels)	Related Works				ARLHT Architecture		
	Ref.	Memory (bits)	DSP Slices	Freq. (MHz)	Memory (bits)	DSP Slices	Freq. (MHz)
333×333	[20]	6,635,520	13	260.06	497,664	0	250.00
512×512	[87]	3,317,760	90	247.53	737,280	0	250.00
	[89]	223,360	0	200.00	442,368	0	250.00
800×600	[88]	262,144	—	—	92,160	0	250.00
1024×768	[21]	1,843,200	0	73.50	645,120	0	250.00
	[13]	3,474,432	8	200.00	774,144	0	250.00
1920×1080	[90]	1,695,744	—	100.00	516,096	0	250.00

The ARLHT architecture consumes less on-chip memory than all of the LHT architectures given in Table 6.10 that use on-chip memory. Only the work presented in [89] uses off-chip memory to store the HPS. The authors report that the memory bandwidth required by their design is 1,172,880 bits for an image resolution of 512×512 pixels. Although the ARLHT requires more on-chip memory than this LHT architecture design, it uses considerably less memory overall. In total, the ARLHT architecture uses approximately 63.32% less memory and can be deployed on FPGA systems that are not connected to external memories. Additionally, the ARLHT has the flexibility to cater for more than one image resolution. Its execution time is not dependent on the bandwidth and speed of the off-chip memory.

Notably, all implementations of the ARLHT achieved a target clock frequency of 250 MHz. It is worth mentioning that the maximum clock frequency of each architecture was not evaluated, which indicates the potential for these architectures to achieve a higher operational clock frequency. The target clock frequency is an improvement

in comparison to all other LHT architectures presented in Table 6.10, except from the implementation described in [20]. Additionally, the ARLHT does not use any DSP48E2 slices as it only requires one vote per edge pixel, which significantly decreases resource requirements in comparison to the parallel LHT and Symmetric LHT. However, this reduction in computational complexity reduces the accuracy of line detection, which was described previously in Section 3.3.1.

The results in this section demonstrate that the ARLHT architecture is resource efficient and can operate at a higher clock frequency than most related works. Although the ARLHT has low memory requirements, the algorithm's limitations should be addressed. These limitations are discussed in the following section.

6.4.5 Limitations and Discussion of Results

The ARLHT is based on the work published by Ser and Siu in [23], who developed an LHT algorithm for a memory-compressed HPS. Their algorithm was applied to an image of 384×256 pixels and used 374,220 bits of memory to store the HPS and RBM. Their algorithm achieved a memory saving of 50% as compared to the standard LHT. The ARLHT can process the same image resolution, and its memory requirements to store the HPS can be computed using (6.2). In total, the ARLHT consumes 311,850 bits to process the same image resolution, which is a memory saving of 16.67% in comparison to the memory-compressed LHT described by Ser and Siu.

There are limitations of the ARLHT that affect its line detection accuracy in digital images and its overall memory requirements to store the HPS. A significant feature of the ARLHT is its ability to exploit the sparsity of peaks in the HPS to reduce its overall memory consumption by an integer factor K_θ. When $K_\theta > 4$ and $N_\theta = 180$, investigations found that extracting peaks from the HPS that correspond to lines in an image is no longer successful. Therefore, the number of angular regions is limited to an integer in the range $1 < K_\theta \leq 4$. This limitation is not explored further in this thesis. However, it is an ideal candidate for future work in this area as it could enable further compression of the HPS and improve memory consumption.

Finally, the memory consumption for the parallel LHT, Symmetric LHT, standard LHT, and ARLHT can be illustrated graphically. Figure 6.18 contains a bar chart of the memory bits for each LHT algorithm across various image resolutions. The ARLHT achieves the lowest memory consumption compared to the other LHT algorithms.

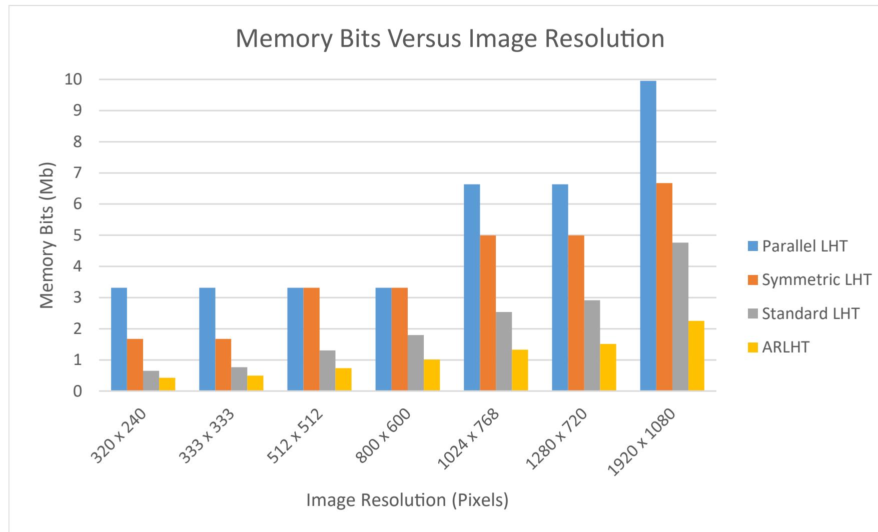


Figure 6.18: A bar chart comparing the number of memory bits required by the parallel LHT, Symmetric LHT, standard LHT, and ARLHT across several image resolutions.

The primary reason for developing the ARLHT algorithm was to achieve sufficient line detection accuracy while reducing the overall memory resources required to store the HPS. The ARLHT reduced the total memory requirements substantially compared to the parallel LHT, Symmetric LHT, and standard LHT. The ARLHT algorithm offers the most effective memory-efficient solution for detecting lines in digital images.

6.5 Conclusion

This chapter has presented a novel algorithm and FPGA architecture for memory-efficient line detection in digital images, named the ARLHT. Initially, an overview of the ARLHT algorithm was described based on the memory-efficient LHT given in [23]. It was demonstrated that the memory requirements of the LHT could be reduced by exploiting the sparsity of peaks within the HPS. Two separate, smaller memories

were employed; a memory-compressed HPS across the θ-axis, and an RBM. It was shown that these memories could be used in conjunction to significantly reduce memory requirements compared to the standard LHT and related works. A crucial component to the success of the ARLHT algorithm was the incorporation of a morphological opening applied to the RBM after voting was complete. This operation enabled the RBM to determine the true orientation of detected peaks in the memory-compressed HPS.

The FPGA implementation for the ARLHT was then described, which included detailed diagrams of the Hough kernel and adjusted orientation architectures. Furthermore, signal flow graphs of the memory-efficient accumulator and RBM circuit designs were presented and their operation was discussed in detail. Finally, the ARLHT architecture was evaluated for its FPGA resource consumption and maximum clock frequency across several image resolutions. For an image of 1920×1080 pixels, the ARLHT architecture consumed 2,248,704 bits of memory (61 BRAMs). The ARLHT architecture saved 52.76% of memory in comparison to the standard LHT.

There are four primary findings and outcomes from the work undertaken in this chapter, which are listed as follows:

1. The ARLHT is the first FPGA architecture that uses compression to reduce the memory requirements of line detection in FPGAs. It was found that the sparsity of peaks could be exploited in the HPS to significantly decrease memory consumption compared to the standard LHT algorithm and parallel LHT architecture.

2. An FPGA architecture of the ARLHT was described that uses two separate memory circuits to store the HPS. These circuits substantially reduce the memory requirements compared to the standard LHT. For an image of 1920×1080 pixels, the ARLHT architecture saves 52.76% of memory resources compared to the standard LHT. Furthermore, when compared with FPGA architectures of the LHT in past works, the ARLHT architecture requires significantly less memory to store the HPS.

3. The ARLHT algorithm uses a lossy compression scheme to store the HPS. This scheme was shown to have sufficient accuracy for up to four angular regions.

4. Small FPGA devices that contain 36 Kb BRAM tiles can target the ARLHT architecture. For instance, the ARLHT architecture can process an edge image of 1920×1080 pixels using only 61 BRAM tiles and does not require any DSP slices. In contrast, the parallel LHT requires 270 BRAM tiles and 89 DSP slices for the same image resolution. While the parallel LHT requires an expensive mid-range or large FPGA, developers can benefit from cheaper FPGAs by using the ARLHT instead. Additionally, small FPGAs are advantageous as they consume less PCB area and have smaller packaging requirements.

Chapter 7

Conclusions

FPGAs are increasingly used to accelerate computer vision algorithms to meet the processing demands of large image resolutions and high video frame rates. Line detection in digital images is essential for many computer vision applications and can be achieved using the well-known LHT algorithm. Although the LHT is very robust to noise and can accurately detect lines in digital images, it is highly computational, and the associated HPS demands significant memory resources.

This thesis has presented a unique evaluation platform for FPGA architectures of the LHT and described two novel LHT algorithms and their memory-efficient FPGA architectures. These architectures target FHD video standards and require substantially fewer memory resources than previously published implementations of the LHT. This chapter reviews the contents of this thesis and presents a summary of its key results. Future work relating to this area of research is also discussed.

7.1 Resume

Chapter 2 described an overview of FPGA and Zynq MPSoC technologies and highlighted the advantages of using PYNQ towards architecture development and validation. Several image processing algorithms were detailed, including local filtering, edge detection, and binary morphological processing, as they were relevant to the novel LHT architectures described in this thesis.

Chapter 3 explored the LHT algorithm and investigated many of its variations described in previously published works. The key findings of the literature review revealed that very few studies investigated the significant memory consumption of the HPS. Applications of the LHT were also explored, which included lane detection for vehicles, and powerline inspection using UAVs. These applications motivated the need for low-latency processing to perform safety-critical tasks.

Chapter 4 described a novel evaluation platform known as the HEP, which enabled the development and validation of LHT architectures. The design of the HEP was presented and the LHT architecture development framework it enabled was explored. A parallel LHT architecture was also described in Appendix B to test the capabilities of the HEP. This architecture successfully demonstrated the features of the HEP, such as its processing time analysis and rapid system integration capabilities.

Chapter 5 presented the Symmetric LHT, which is a novel architectural optimisation of the LHT. The Symmetric LHT decreased the memory allocation of the HPS in FPGA devices by exploiting a symmetrical coordinate system in the spatial image domain. A bit-packed accumulator was used to store the votes for two angles in θ into the same location in memory, reducing the overall memory requirements in FPGA devices compared to the parallel LHT. The FPGA architecture of the Symmetric LHT was described and its performance in terms of resource consumption and timing closure was presented. The Symmetric LHT architecture was also compared to previously published implementations of the LHT to highlight its memory efficiency.

Chapter 6 detailed the ARLHT, which is a new algorithmic modification of the LHT based on the work presented in [23]. The operation of the ARLHT was described, where the size of the HPS was decreased along the θ-axis. The ARLHT's voting scheme was also demonstrated using two memories (a compressed HPS and RBM) in conjunction to significantly reduce memory utilisation compared to the standard LHT. A morphological opening operation was applied to the RBM, which was crucial to the success of the ARLHT. An FPGA architecture of the ARLHT was presented and compared to previously published LHT implementations in the literature to highlight its memory efficiency. Additionally, the limitations of the ARLHT were also discussed.

7.2 Discussion of Results

The literature review presented in Section 3.3 details variations of the LHT algorithm and related FPGA implementations. Key findings from this review revealed that few investigations improve the memory consumption of the HPS in FPGA devices. The LHT is a widely adopted algorithm for detecting lines in digital images, and FPGAs are a popular choice to accelerate the algorithm for embedded applications. Due to these reasons, the memory-efficient FPGA implementation of the LHT is an interesting problem that this thesis set out to address. This section presents a summary of the key results of this thesis.

In Chapter 4, a novel evaluation platform for FPGA architectures of the LHT, named the HEP, was presented. The HEP uses MathWorks *HDL Coder*, the PYNQ software framework, and the Vivado Design Suite to provide a rapid development environment that offers repeatable results. The processing time of an LHT architecture can easily be measured using the HEP, and its visualisation and analysis tools can be used to inspect the output HPS. To the author's knowledge, this development environment is the first to combine PYNQ and MathWorks *HDL Coder*. A significant achievement of this work was the open-sourcing of the HEP, now a software tool that other researchers can download freely online. The HEP allows LHT architecture designs to be compared fairly and with research integrity. Additionally, the HEP was used to rapidly design and evaluate the FPGA architectures of the parallel LHT, Symmetric LHT, and ARLHT, demonstrating its hardware validation and design evaluation capabilities.

To test the operation and evaluate the capabilities of the HEP, a parallel LHT architecture based on the work presented in [87] was developed (see Appendix B). This parallel LHT design is functionally equivalent to the standard LHT and was designed to process an image of 1920×1080 pixels. The parallel LHT architecture was implemented and deployed on the XCZ7UEV-2E device using the HEP. The architecture was successfully hardware validated on the physical target device using two candidate test images. The architecture achieved a maximum clock frequency of 200 MHz and required 89 DSP48E2 slices and 270 BRAM tiles. The parallel LHT consumed 86.54% of all

BRAM tiles on the XCZ7UEV-2E device, which is a considerable amount of memory. Many embedded applications given in Section 3.4 could leverage a memory-efficient implementation of the LHT for two primary reasons. Firstly, a financially cheaper FPGA could be selected instead of the XCZ7UEV-2E device, which would decrease the overall cost of the system. A financially inexpensive FPGA could also significantly reduce the cost of embedded solutions that undergo mass manufacture. Secondly, the LHT could be implemented in resource-constrained environments where optimising BRAM tile consumption is a critical design parameter. A memory-efficient LHT would be able to co-exist alongside other hardware accelerators on an FPGA simultaneously.

In Chapter 5, the Symmetric LHT is presented, which optimises the memory requirements of the accumulator memory by exploiting spatial domain symmetry and bit-packing techniques. The Symmetric LHT architecture was developed using the HEP and deployed and validated on the target XCZ7UEV-2E device. The architecture consumed 181 BRAM tiles to store the HPS for an image of 1920×1080 pixels. This memory consumption is a significant improvement compared to the parallel LHT architecture, as it uses 32.96% less memory. The Symmetric LHT architecture also achieved a maximum clock frequency of 205 MHz. The HEP reported the time to process one image using the Symmetric LHT architecture as 12.06 ms, which corresponds to 82.92 fps. The Symmetric LHT processing time results and overall memory consumption are better than the parallel LHT architecture. Additionally, the frame rate is suitable for the FHD video standard.

The Symmetric LHT was also compared to previously published implementations of the LHT and compared favourably in terms of memory consumption and processing time. The memory requirements of the Symmetric LHT were also compared with that of the standard LHT across various image resolutions. Although the Symmetric LHT has lower memory requirements than the parallel LHT and other related works, the memory efficiency of the bit-packed accumulator could be improved compared to the standard LHT. For instance, the standard LHT uses 4,790,640 bits to store the HPS for an image of 1920×1080 pixels. For the same image resolution, the Symmetric LHT requires 6,672,384 bits for the accumulator memory, inefficiently allocating 140.16% of

memory resources. The overall impact of using bit-packing techniques reduced the total BRAM requirements of the HPS when compared to the parallel LHT and previously published works. However, the Symmetric LHT still consumed a significant amount of memory to store the HPS for various image resolutions, demanding FPGAs with large quantities of BRAM tiles. It was concluded that the memory consumption of the HPS could be improved further by exploring compression techniques.

In Chapter 6, the ARLHT and its FPGA architecture are described. The ARLHT is an algorithmic modification of the standard LHT that aims to reduce the memory requirements of the HPS. The HEP was used to develop the FPGA architecture of the ARLHT, which was implemented and successfully hardware validated using two candidate test images on the XCZ7UEV-2E device. For an image of 1920×1080 pixels, the ARLHT architecture only required 61 BRAM tiles to store the HPS. Compared to the memory consumption of the standard LHT for the same image resolution, the ARLHT saves 52.76% of memory. The memory consumption of the ARLHT also compares favourably to previously published implementations of the LHT. The ARLHT architecture achieved a target clock frequency of 200 MHz and was reported by the HEP to process an image in 12.37 ms. This processing time corresponds to 80.84 fps, which is suitable for the FHD video standard.

At the time of writing, the XCZ7UEV-2E device costs approximately £3,618 to purchase online at the DigiKey store (a supplier trading in the United Kingdom) [117]. The ARLHT significantly impacts the total memory consumption of the HPS. Financially inexpensive FPGAs with relatively few BRAM tiles, such as the XC7A50T device [118], can apply the ARLHT to a digital image of 1920×1080 pixels. The XC7A50T device only contains 75 BRAM tiles and 120 DSP slices and currently costs £55.72 on the DigiKey store [119]. This FPGA is significantly cheaper than the XCZ7UEV-2E device (although the XC7A50T FPGA does not contain a PS). The XC7A50T FPGA cannot apply the Symmetric LHT or parallel LHT to an image of 1920×1080 pixels as it does not contain sufficient BRAM resources to store the HPS.

The results presented in this thesis have demonstrated the memory-efficient implementation of the LHT in FPGA devices. By reducing the memory requirements of the HPS, the LHT can be implemented in memory-constrained FPGA systems. Many embedded applications may benefit from low-memory LHT architectures, as smaller FPGA devices with fewer BRAM tiles can be selected for the design of image processing and computer vision systems.

7.3 Key Conclusions

The primary objective of this research was to determine whether it is possible to reduce the memory consumption of the HPS so that the LHT can be implemented using small FPGA devices. The key findings presented throughout this thesis have demonstrated that it is possible to reduce the memory requirements of the HPS by using bit-packing techniques and data compression. The Symmetric LHT used spatial domain symmetry and bit-packing to reduce FPGA resource consumption, while the ARLHT exploited the sparsity of peaks in the HPS to compress data and significantly decrease FPGA memory requirements. Furthermore, FPGA architecture designs of these techniques were developed using the novel HEP and were successfully validated on the physical target hardware, confirming their functional operation.

The Symmetric LHT produces the same results as the parallel LHT when configured to operate using the same set of parameters, i.e. δ_θ, δ_ρ, N_θ, and N_ρ. Developers can leverage the memory efficiency and improved DSP slice consumption offered by the Symmetric LHT in their embedded system designs. In particular, the Symmetric LHT can replace implementations of the parallel LHT to improve FPGA resource consumption and allow developers to select smaller FPGA devices. There are only two issues that developers should be aware of when using the Symmetric LHT architecture. Firstly, the input image must have an even number of rows and columns. Secondly, there must be an even number of angles in θ as this allows resource-sharing techniques to be used, which improves the DSP slice consumption of the architecture design.

The ARLHT offers significant improvements in memory consumption compared to the standard LHT, parallel LHT, and Symmetric LHT. Developers should use the

ARLHT to detect lines in digital images for memory-constrained FPGA systems. Although the ARLHT consumes very few FPGA resources, the algorithm does have weaknesses that should be addressed. The ARLHT applies one vote to the HPS per edge pixel, which is similar to the Gradient LHT when $\lambda = 0$. This voting scheme can reduce the sensitivity of line detection compared to the standard LHT, which is caused by edge pixels that are not entirely collinear in the spatial domain. Furthermore, the accuracy of line detection can decrease when the number of angular regions used by the ARLHT is greater than four. Note that the accuracy and capabilities of the ARLHT algorithm have been analysed and evaluated in recent literature [120].

The original research in this thesis has demonstrated that engineers and developers can leverage memory-efficient LHT architectures in their embedded vision applications. These architectures can target small FPGAs, which have lower device costs, smaller PCB areas, and improved energy efficiency compared to large FPGAs. Optimised architecture designs also offer developers additional FPGA resources to target other algorithms and tasks using the same bitstream.

7.4 Future Work

There are various avenues of future work for this research area that would be interesting to explore. These are listed below:

- In Chapter 4, the HEP evaluates an LHT architecture by transferring image data between the Zynq MPSoC's PS and PL using an AXI DMA. It would be useful to extend the HEP's functionality so that the user can interface a video camera to the input of their custom LHT architecture. Similarly, the output of their architecture could be connected to a display interface such as HDMI. This functionality would allow the user to stream video data through LHT architecture designs.

- The Symmetric LHT architecture in Chapter 5 uses two separate memories to store the votes for θ_0 and $\theta_{N_\theta/2}$. It would be interesting to explore whether the votes for these angles could be bit-packed into the same memory locations. This

optimisation would improve the BRAM tile allocation for most image resolutions, further reducing memory requirements.

- In Chapter 5, the spatial domain symmetry exploited by the Symmetric LHT is only performed across the y-axis. Although external memory may be required to buffer the input image, the symmetry across the x-axis should also be investigated. If x-axis symmetry could be achieved, the resulting FPGA architecture can be used to efficiently apply the LHT to spectrogram plots, enabling the detection of chirps for radar applications.

- If there are more than four angular regions, the ARLHT architecture in Chapter 6 is unable to reliably detect peaks in the HPS. To improve the ARLHT, a new method of peak separation is required to suppress the spurious detection of lines and further reduce memory consumption. Additionally, the ARLHT should undergo further analysis such as that performed in [120] to determine its performance in terms of line detection accuracy.

- Finally, it would be interesting to combine the ARLHT with work presented in [23], wherein the ρ-axis of the HPS was reduced by partitioning the input image into subregions. The resulting LHT algorithm would be very efficient in terms of memory utilisation. However, the algorithm would exhibit similar limitations as the ARLHT, described in Section 6.4.5. These limitations require further investigation.

Appendix A

CORDIC

COordinate Rotation DIgital Computer (CORDIC) is an iterative hardware architecture initially developed by Volder [72] that uses shift and add operations to implement trigonometric functions efficiently. Walther [73] extended the technique to implement multiplication, division, hyperbolic functions, and logarithmic functions. Only the trigonometric derivation, also known as Circular CORDIC, is required in this thesis for calculating the gradient orientation of a greyscale image (see Section 2.6.7).

The Circular CORDIC algorithm operates on the principle of iteratively rotating an input vector (x_0, y_0) by a set of angles θ_k to produce an output vector (x_K, y_K), where k maintains track of the current iteration and K is the total number of iterations. Figure A.1 presents an example that demonstrates three iterations of the CORDIC algorithm, where a vector is rotated on the Cartesian plane.

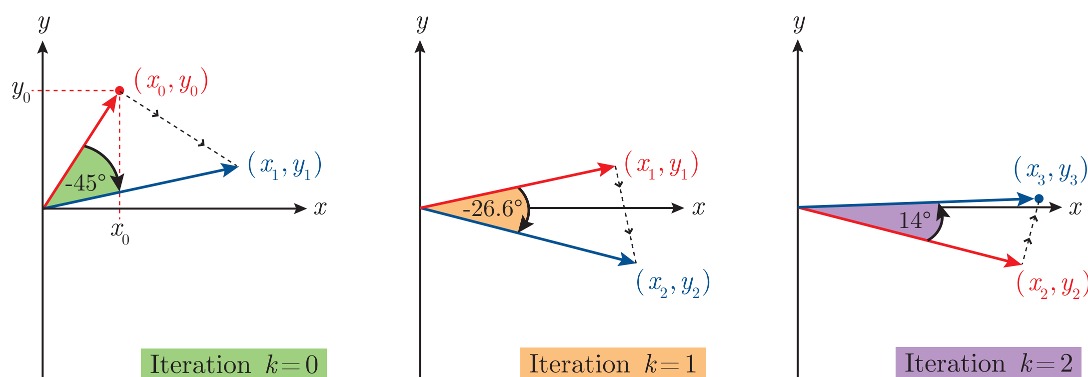


Figure A.1: An example of rotating a vector (x_0, y_0) three times on the Cartesian plane.

Circular CORDIC is an effective technique to compute trigonometric functions such as $\cos(\theta)$ and $\sin(\theta)$ on an FPGA device. The algorithm can also calculate the inverse tangent and magnitude of an input vector. The following sections introduce the Circular CORDIC algorithm and include discussions on the scaling factor, the cumulative angle, and the region of convergence.

A.1 The Circular CORDIC Algorithm

The Circular CORDIC algorithm operates by iteratively rotating a vector on the Cartesian plane to converge on a result. A vector rotation on the Cartesian plane can be mathematically expressed as

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta_k) & -\sin(\theta_k) \\ \sin(\theta_k) & \cos(\theta_k) \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix}. \tag{A.1}$$

The relationship above can be rewritten by taking a factor of $\cos(\theta_k)$ and substituting in $\tan(\theta_k) = \sin(\theta_k)/\cos(\theta_k)$, which gives

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \cos(\theta_k) \begin{bmatrix} 1 & -\tan(\theta_k) \\ \tan(\theta_k) & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix}. \tag{A.2}$$

The remaining $\cos(\theta_k)$ term in (A.2) can be dropped to simplify the implementation of the CORDIC equations. Later in Section A.2, the $\cos(\theta_k)$ term is reintroduced to prevent errors. The relationship now becomes

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & -\tan(\theta_k) \\ \tan(\theta_k) & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix}. \tag{A.3}$$

To simplify (A.3) for hardware implementation, only values of $\tan(\theta_k)$ that are consecutive negative powers of two will be selected to rotate the vector (x_k, y_k). Restricting the angle of rotation in this way allows the $\tan(\theta_k)$ terms to be converted from multiplications, to simple bit-shifts to the right. Table A.1 presents the first four rotation angles and their corresponding values of $\tan(\theta_k)$.

Table A.1: The rotation angles (degrees) for the first four iterations of the CORDIC algorithm. The rotation angles are given to four decimal places.

Iteration Number (k)	**Rotation Angle** (θ_k°)	$\tan(\theta_k) = 2^{-k}$
0	45.0000	1
1	26.5651	0.5
2	14.0362	0.25
3	7.1250	0.125

Figure A.2 presents an example of the first three CORDIC iterations using the rotation angles given in Table A.1. The first iteration rotates the input vector by 45°, the second iteration rotates the vector by 26.5651°, and the third iteration rotates the vector by 14.0362°. The rotation angle becomes smaller after each iteration, allowing the CORDIC algorithm to converge on a solution.

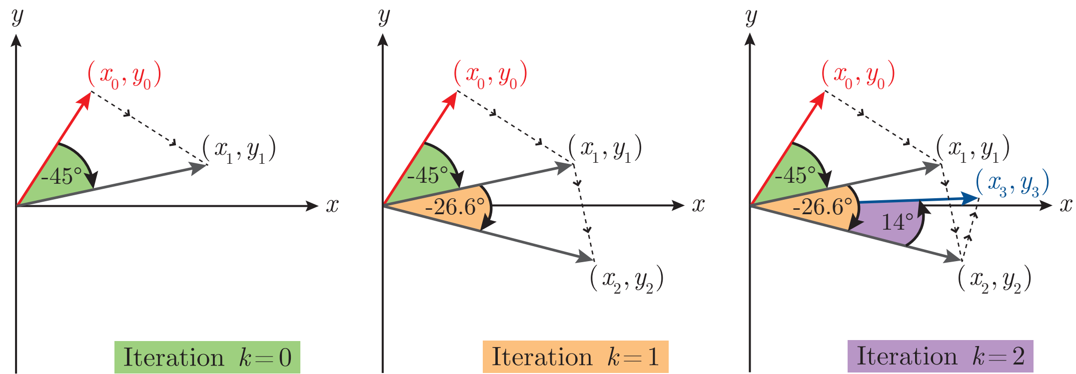


Figure A.2: Plots of the first three CORDIC iterations using the rotation angles given in Table A.1. The rotation angle is given to one decimal place and becomes smaller after each iteration. The scaling factor (discussed in Section A.2) is ignored.

The rotation angles given in Table A.1 are all positive values. Negative rotation angles are introduced by using a decision variable d_k that allows the CORDIC algorithm to rotate an input vector in a clockwise or anticlockwise direction. The value of d_k is +1 for an anticlockwise rotation and -1 for a clockwise rotation. The relationship in (A.3) can now be rewritten as

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & -d_k 2^{-k} \\ d_k 2^{-k} & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix}. \tag{A.4}$$

The decision variable d_k changes the sign of the term 2^{-k} to allow the CORDIC algorithm to converge on a solution. The value of d_k is set depending on the operational mode of CORDIC, which is discussed further in Section A.5.

A.2 The Scaling Factor

A side-effect of dropping the $\cos(\theta)$ term from the CORDIC equation in (A.3) is that the input vector will increase in magnitude/length after each CORDIC iteration. This problem is normally referred to as vector growth and can be demonstrated by plotting the first three iterations of the CORDIC algorithm, as given in Figure A.3. Notice that the input vector grows in magnitude after each iteration.

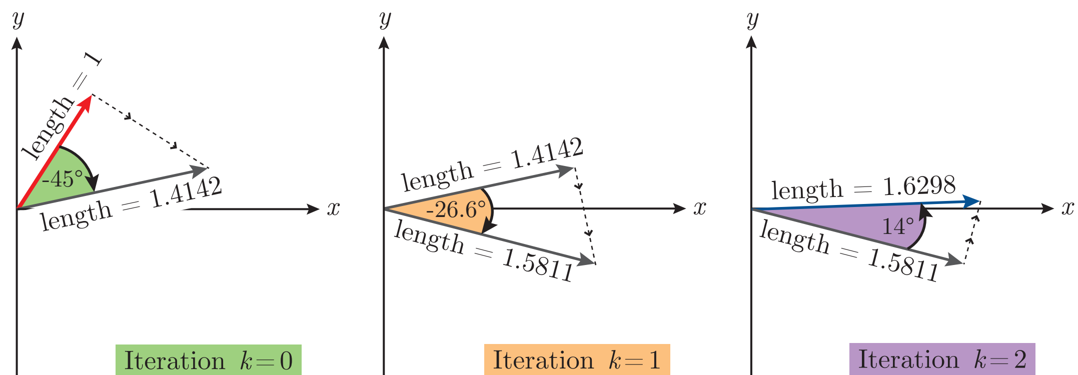


Figure A.3: These plots demonstrate vector growth after three CORDIC iterations. The rotation angles are given to one decimal place and the vector length is given to four decimal places.

Table A.2 presents the vector growth for the first four CORDIC iterations. Notice that the vector growth tends towards one after each iteration due to the rotation angle progressively decreasing in size. The vector growth after each iteration is given by $1/\cos(\theta_k)$, where θ_k is the angle of rotation.

Table A.2: The vector growth for the first four iterations of the CORDIC algorithm. The rotation angles (degrees) and vector growth are given to four decimal places.

Iteration Number (k)	**Rotation Angle** ($\theta_k{}^\circ$)	**Vector Growth**
0	45.0000	$1/\cos(45.0000^\circ) = 1.4142$
1	26.5651	$1/\cos(26.5651^\circ) = 1.1180$
2	14.0362	$1/\cos(14.0362^\circ) = 1.0308$
3	7.1250	$1/\cos(7.1250^\circ) = 1.0079$

After applying K rotations on an input vector, the overall vector growth G_{K-1} is a constant that can be calculated by computing the product of $1/\cos(\theta_k)$ after each consecutive rotation. This is mathematically expressed as

$$G_{K-1} = \prod_{k=0}^{K-1} \frac{1}{\cos(\theta_k)}. \tag{A.5}$$

Note that when K approaches infinity, the vector growth converges and is approximately 1.6467. To compensate for vector growth, a scaling factor can be applied to the input or output of the CORDIC processor. FPGA architectures usually apply the scaling factor using a constant multiplication, which requires very few arithmetic resources. The scaling factor is defined as the reciprocal of the overall vector growth, which is given as $1/G_{K-1}$. Later in Section A.5, an example is presented of an FPGA architecture that uses a constant multiplication to apply the scaling factor to the output of a CORDIC processor.

A.3 The Circular CORDIC Equations

An additional register z_k is usually introduced to maintain the cumulative angle between rotations. The rotation angle for a CORDIC iteration can be computed using $d_k \tan^{-1}(2^{-k})$, where d_k determines the direction of rotation. The rotation angle is subtracted from z_k when rotating anticlockwise and added to z_k when rotating clockwise. This operation is expressed mathematically as

$$z_{k+1} = z_k - d_k \tan^{-1}(2^{-k}). \tag{A.6}$$

Linear equations can be also be used to represent (A.4) as follows.

$$x_{k+1} = x_k - d_k 2^{-k} y_k$$
$$y_{k+1} = y_k + d_k 2^{-k} x_k \tag{A.7}$$

The scaling factor is not included in the above equations as the overall scaling factor $1/G_{K-1}$ is usually applied after all CORDIC iterations are completed.

A.4 Region of Convergence

Circular CORDIC has limitations on the range of angles that it will successfully converge upon. The angle of rotation becomes smaller after each iteration, which causes the cumulative angle to converge to an upper or lower limit. Figure A.4 demonstrates the first six CORDIC rotations on the Cartesian plane.

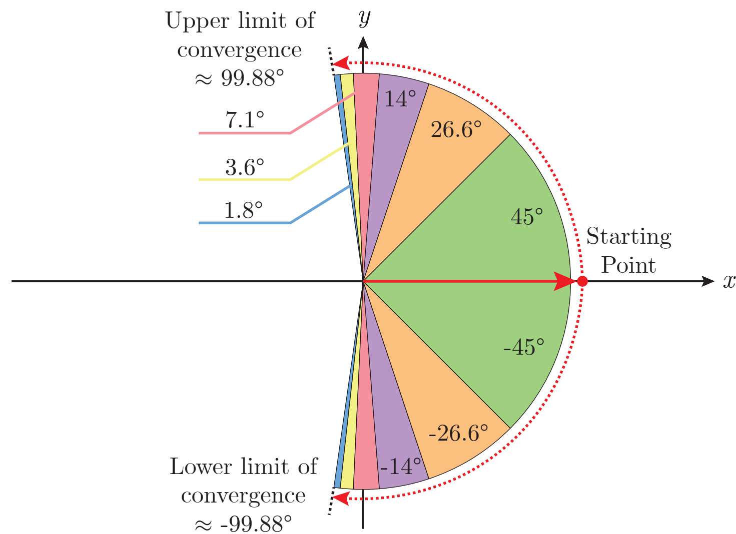


Figure A.4: A plot demonstrating the first six CORDIC rotations tending towards a limit in the anticlockwise and clockwise directions.

The cumulative angle converges to a limit in the anticlockwise and clockwise directions, which can be computed using

$$\sum_{k=0}^{\infty} \tan^{-1}(2^{-k}) \approx 99.88°. \tag{A.8}$$

CORDIC is unable to converge on angles that are larger than $\pm 99.88°$. This issue can be addressed by adjusting the input vector (x_0, y_0) by $\pm 90°$ and then correcting the angle of rotation, z_K, after the CORDIC operation is complete. These operations are referred to as quadrant mapping and demapping and are only required when the designer is aware that the angle of convergence is larger than 99.88°. An input vector (x_0, y_0) can be rotated by multiples of 90° using a simple technique. An example is presented in (A.9) that rotates a vector by 90° in the clockwise direction.

$$\begin{bmatrix} y_0 \\ x_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} -x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} \cos(-90°) & -\sin(-90°) \\ \sin(-90°) & \cos(-90°) \end{bmatrix} \begin{bmatrix} -x_0 \\ y_0 \end{bmatrix} \tag{A.9}$$

Similarly, a vector can be rotated by 90° in the anticlockwise direction as follows.

$$\begin{bmatrix} y_0 \\ -x_0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} -x_0 \\ -y_0 \end{bmatrix} = \begin{bmatrix} \cos(90°) & -\sin(90°) \\ \sin(90°) & \cos(90°) \end{bmatrix} \begin{bmatrix} -x_0 \\ -y_0 \end{bmatrix} \tag{A.10}$$

For inspection purposes, the rotation examples in (A.9) and (A.10) can be illustrated on a plot, as shown in Figure A.5. Notice that each input vector initially lies outside the region of convergence for Circular CORDIC. Each vector is then rotated by 90° into the region of convergence.

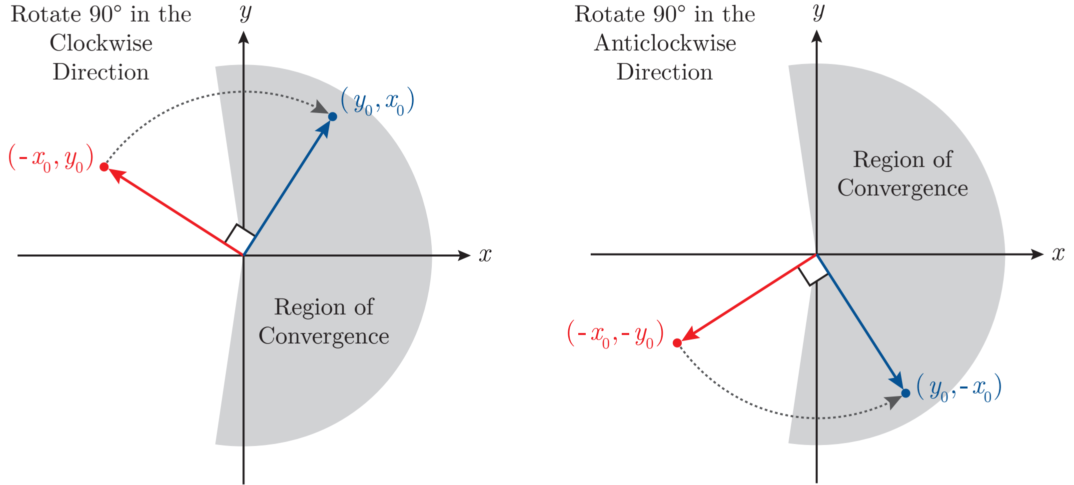


Figure A.5: Two plots demonstrating the rotation of a vector into the region of convergence for Circular CORDIC. The left plot illustrates a vector rotation by 90° in the clockwise direction, while the right plot shows a vector rotation by 90° in the anticlockwise direction.

Quadrant mapping can be implemented on an FPGA by first establishing if the x_0 and y_0 inputs are positive or negative and subsequently rotating the vector as required. For example, if x_0 and y_0 are negative, the input vector should be rotated 90° in the anticlockwise direction. Similarly, if x_0 is negative and y_0 is positive, the input vector should be rotated 90° in the clockwise direction.

A.5 Vectoring Mode

Circular CORDIC has two modes of operation known as rotation and vectoring mode. The following discussion will be limited to vectoring mode as it is the only mode relevant to the work in this thesis. To initialise the CORDIC processor for vectoring mode, the direction of rotation d_k should reduce y_k towards zero after each iteration, as below.

$$d_k = -\text{sign}(y_k). \tag{A.11}$$

The input vector (x_0, y_0) is given in Cartesian coordinates and the angle of rotation z_0 is set to zero. In this configuration, the CORDIC processor will perform pseudo-rotations with the aim of rotating the input vector towards the x-axis, as shown in Figure A.6.

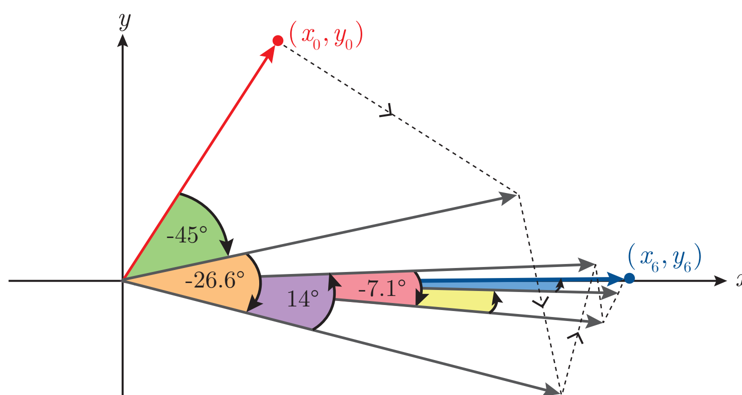


Figure A.6: An example of Circular CORDIC operating in vectoring mode. The input vector is rotated towards the x-axis.

The z_k register stores the cumulative angle of rotation after each iteration. The total angle of rotation z_K can be found after all iterations have been performed. The final value of x_K is the magnitude of the input vector (since $y_k \to 0$), which will need

to be scaled by $1/G_{K-1}$. The theoretical response of Circular CORDIC operating in vectoring mode after K pseudo-rotations is expressed using the following equations.

$$\begin{aligned} x_K &= G_{K-1}\sqrt{x_0^2 + y_0^2} \\ y_K &= 0 \\ z_K &= z_0 + \tan^{-1}\left(\frac{y_0}{x_0}\right) \end{aligned} \tag{A.12}$$

Figure A.7 presents an architecture for a Circular CORDIC processor, which contains optional quadrant mapping and demapping stages at the input and output of the processor, respectively.

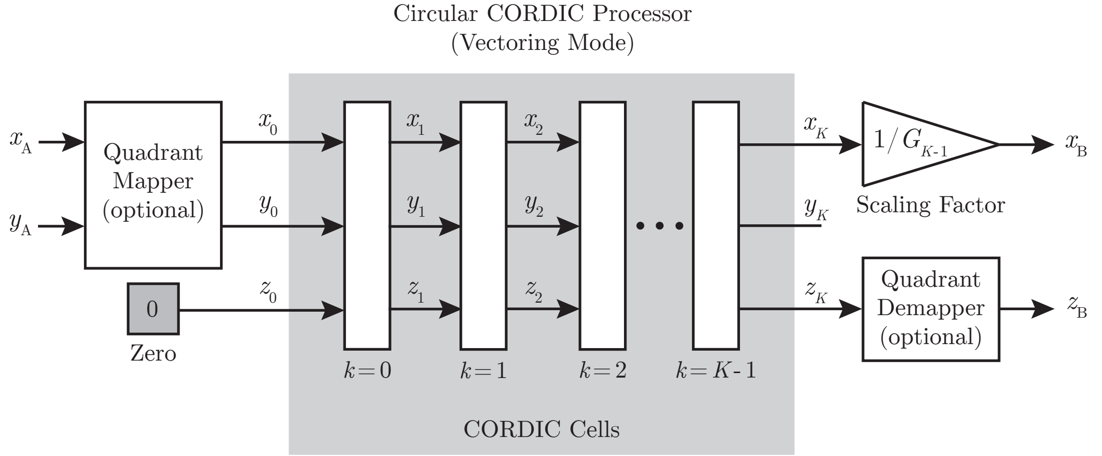


Figure A.7: Hardware architecture of Circular CORDIC operating in vectoring mode.

The input to the quadrant mapper is a vector denoted as (x_A, y_A). The quadrant mapper rotates the vector by ±90° if it is outside the region of convergence, creating a new vector (x_0, y_0). The new vector is input into the CORDIC processor, which contains K CORDIC pseudo-rotations. Notice that the cumulative angle of rotation is also computed between each CORDIC pseudo-rotation. The input cumulative angle register is labelled z_0, while the output register is labelled z_K. The quadrant demapper is used to correct the angle z_k as required, giving the final rotation angle z_B. The last stage of the architecture removes the scaling introduced by the CORDIC pseudo-rotations by multiplying x_K by $1/G_{K-1}$ to obtain x_B.

Each CORDIC pseudo-rotation block in Figure A.7 is constructed of an efficient shift and add architecture design. The pseudo-rotation block implements the system of linear equations for Circular CORDIC given in (A.6) and (A.7). Figure A.8 presents a signal flow graph for this hardware architecture. Notice that the CORDIC cell architecture uses three additions, two bit-shift operations, and a LUT to store the value of $\tan^{-1}(2^{-k})$. The reason for applying the scaling factor after the CORDIC processor is to reduce the complexity of the cell architecture.

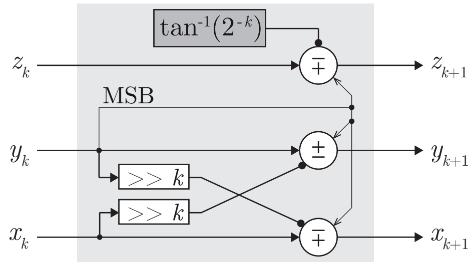


Figure A.8: A hardware architecture of a Circular CORDIC cell for vectoring mode operation.

A.6 Example: Circular CORDIC in Vectoring Mode

Circular CORDIC can be used to compute the magnitude and orientation of a vector. For example, consider the vector $(-4, 3)$. Initially, the quadrant mapping technique described in Section A.4 is required to rotate the vector by 90° in the clockwise direction so that it is within the region of convergence for Circular CORDIC. The vector is equal to $(3, 4)$ after quadrant mapping. The new vector is input into a Circular CORDIC processor operating in vectoring mode. In this example, there are eight CORDIC iterations, which are computed as shown in Table A.3. Notice that the input value of z_0 is zero, so that the value of z_8 at the output of the CORDIC processor is equal to the total angle of rotation.

The output angle z_8 is quadrant corrected by rotating it in the anticlockwise direction. This operation is performed by adding 90° to z_8, which gives $53.5330° + 90° = 143.5330°$. Meanwhile, the scaling factor can be applied to x_8 to reveal the true magnitude of the input vector, i.e. $8.2335/G_{K-1} = 5$.

Table A.3: Example of Circular CORDIC operating in vectoring mode using eight iterations. Each value is given to four decimal places.

k	$\theta_k{}^\circ$	x_k	y_k	$z_k{}^\circ$
0	45.0000	3.0000	4.0000	0.0000
1	26.5651	7.0000	1.0000	45.0000
2	14.0362	7.5000	-2.5000	71.5651
3	7.1250	8.1250	-0.6250	57.5288
4	3.5763	8.2031	0.3906	50.4038
5	1.7899	8.2275	-0.1221	53.9801
6	0.8952	8.2314	0.1350	52.1902
7	0.4476	8.2335	0.0064	53.0854
8	0.2238	8.2335	-0.0579	53.5330

A.7 Summary

This appendix has reviewed the Circular CORDIC algorithm. In particular, the scaling factor, angle accumulator, region of convergence, and vectoring mode of Circular CORDIC were discussed. A hardware architecture and simple example for Circular CORDIC operating in vectoring mode was presented.

Appendix B

Parallel LHT Implementation

This appendix details the design of a parallel LHT architecture for implementation on an FPGA device. The architecture design is based on the work presented in [87] and was developed using MathWorks *HDL Coder* and Simulink. Initially, this appendix summarises Mathworks *HDL Coder* compatible blocks and fixed-point data types. Then, the parallel LHT architecture is presented using Simulink diagrams. Lastly, an LHT software model is presented, which is used for architecture validation.

The author of this thesis has made the best effort to prepare the Simulink diagrams in this appendix for electronic viewing and printing on physical media for reading. However, please note that some diagrams may only be readable using a high-resolution Portable Document Format (PDF) of this thesis. Alternatively, all Simulink models and diagrams can be accessed online in [106] for inspection and analysis.

B.1 MathWorks *HDL Coder* Blocks

MathWorks *HDL Coder* contains a wide variety of HDL-compatible blocks for developing FPGA architecture designs. The work in this thesis only uses a subset of these blocks, as shown in Figure B.1. Each block has been allocated a colour to improve the inspection of Simulink model designs that are presented later in this appendix. Also, note that each block has a number, which is used below to refer to an individual block and summarise its functionality.

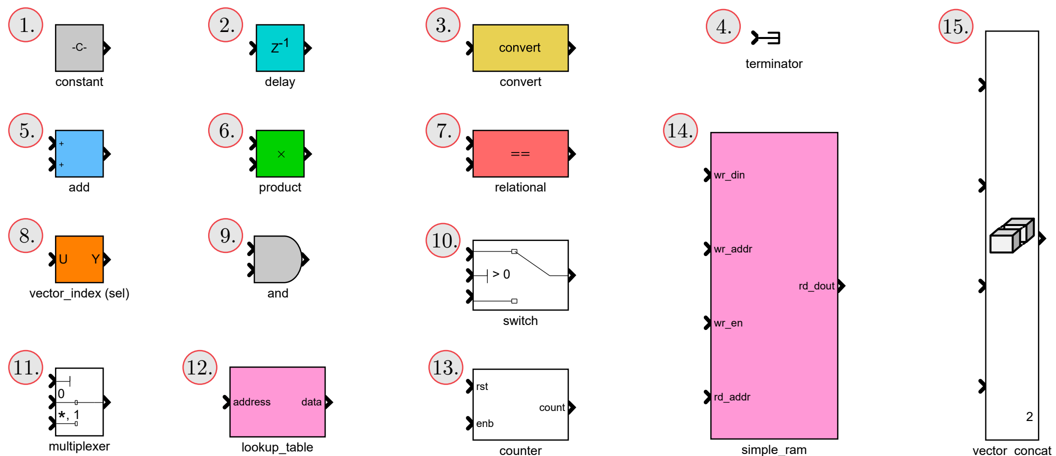


Figure B.1: This diagram presents a subset of MathWorks *HDL Coder* blocks used to develop the LHT architecture designs presented in this thesis.

1. *constant* — The constant block generates a signal that remains at a constant value during architecture operation. The user can configure the constant block to generate a scalar or vector signal.

2. *delay* — This block delays the input signal by a fixed number of clock cycles. The user can specify the delay length in the block's configuration window.

3. *convert* — The convert block is also referred to as a data type conversion. This block converts the data type of an input signal to a user-specified data type.

4. *terminator* — The terminator is used at the output ports of blocks that are not required in the architecture design. Its purpose is to cap an unused output port, which keeps the Simulink design tidy.

5. *add* — This block can perform an add or subtract operation on scalar and vector input signals.

6. *product* — The product block can multiply two input signals together. This block can operate on scalar and vector inputs.

7. *relational* — The relational block implements a relational operator that can be used on scalar and vector input signals. These operators include $==$, $\sim=$, $<$,

<=, >=, and >. See the MathWorks documentation for more information [113].

8. *vector_index (sel)* — This block is also referred to as a selector, or *sel.* The purpose of the selector is to index elements from an input vector.

9. *and (logical)* — The *and* block is an example of a logical block that can be found in the *HDL Coder* block set. Other logical blocks include OR, NAND, NOR, XOR, NXOR, and NOT, which are all common logic functions.

10. *switch* — The switch block has three inputs and one output. The first input and third input are fed to the output based on the value of the second input. The criteria for passing the first or third input is set by the user in the block's configuration window.

11. *multiplexer* — The multiplexer block is also known as a multiport switch. The first input of this block is a control signal, which determines the input signal that is sent to the output port.

12. *lookup_table* — This block implements a one-dimensional function and is commonly abbreviated as LUT. The input to the LUT block is an address, which is used to index a memory location and output the data stored at that location.

13. *counter* — Counter blocks are used to implement a free-running or count-limited counter in the architecture design.

14. *simple_ram* — The simple RAM block (also known as a simple dual port RAM) implements a simultaneous read-and-write RAM. In this thesis, the simple RAM block is used to instantiate a BRAM primitive in the FPGA design.

15. *vector_concat* — This block concatenates two or more vectors together.

Many of the blocks described above will be used in architecture designs to manipulate and process fixed-point data types. MathWorks *HDL Coder* has a fixed-point data type notation that is documented in [121]. Simulink designs in this thesis also use vector signals to simplify FPGA architecture development. Figure B.2 demonstrates how fixed-point data types and vector signals are presented in Simulink.

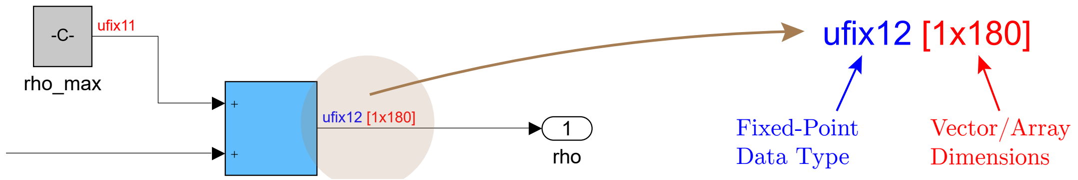


Figure B.2: A diagram illustrating the Simulink convention for displaying a fixed-point data type and dimension of a signal.

The above convention for displaying the fixed-point data type and dimension of a signal will be used in Simulink diagrams throughout the remainder of this appendix. The parallel LHT architecture will now be presented in the following sections.

B.2 Architecture Overview

The parallel LHT architecture will be developed using the HEP (see Chapter 4). Figure B.3 contains a diagram that presents an overview of the entire FPGA design, which includes the LHT architecture and HEP infrastructure.

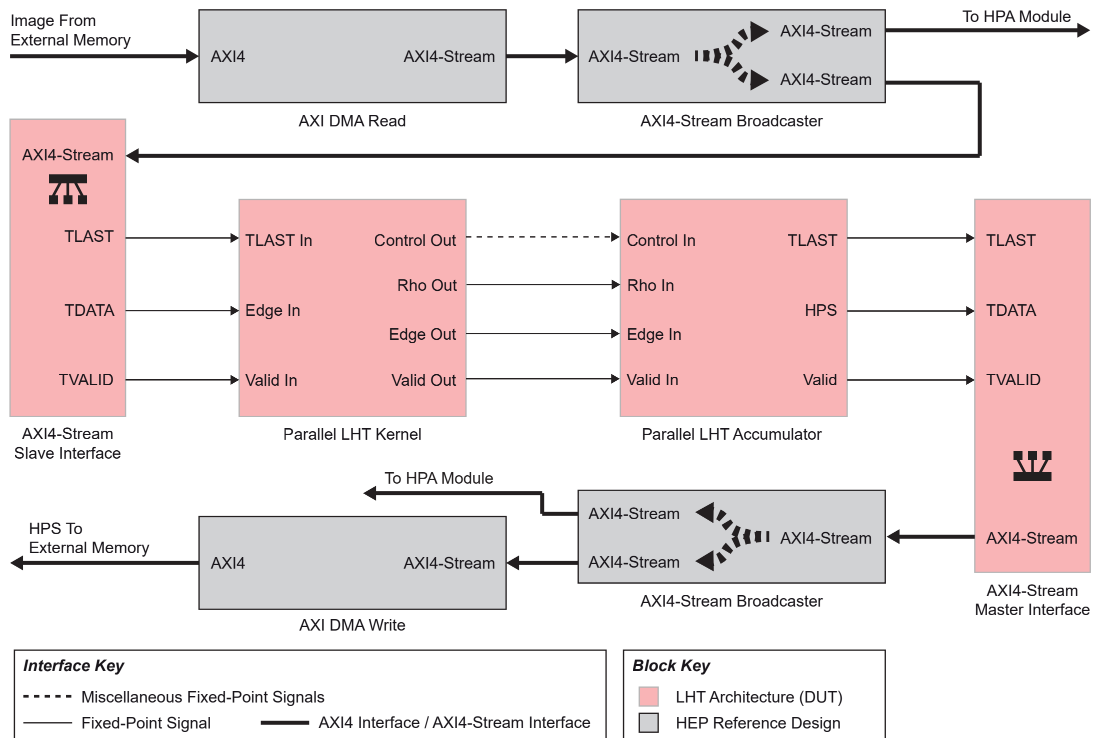


Figure B.3: An overview diagram of the parallel LHT architecture and HEP infrastructure.

The parallel LHT design is separated into two main blocks, the *Parallel LHT Kernel* and *Parallel LHT Accumulator*. The Simulink systems for each of these blocks will be presented in this appendix. Note that discussion of the parallel LHT design will be kept to a minimum, and only information relevant to the Simulink design will be discussed. See [87] for more information on the parallel LHT architecture.

The HEP was used to develop the parallel LHT architecture design. The Simulink template presented in Figure 4.8 was used as a starting point for this work. The contents of the DUT block were modified to contain the subsystems presented in Figure B.4. The wordlength of the TDATA signal at the input and output of the DUT must be 32 bits. Therefore, a *convert* block was used to appropriately modify the fixed-point data type at the accumulator output.

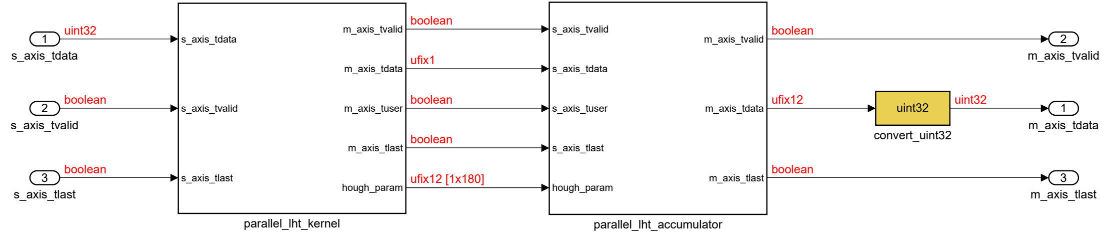


Figure B.4: A Simulink system presenting the contents of the DUT block, which contains the parallel LHT kernel and accumulator. The *convert* block is used to ensure the TDATA signal at the output of the DUT is 32 bits, which is compliant with the HEP design methodology.

The parallel LHT design is configured to process an image containing 1920×1080 pixels, where $\delta_\theta = 1°$, $\delta_\rho = 1$, $N_\theta = 180$, and $N_\rho = 2204$. The input to the architecture is an edge image that uses 1 bit to represent active edge pixels. The least significant bit of the architecture's 32 bit input will contain the edge pixel information.

B.3 Parallel LHT Kernel

Figure B.5 presents the Simulink system for the *parallel_lht_kernel* block. The diagram contains two subsystem blocks named *coordinate_counters* and *parallel_kernel*. Additionally, there are two relational operators that detect the first pixel in an image. The *bitslice* block at the top of the diagram extracts the edge pixels from the input signal.

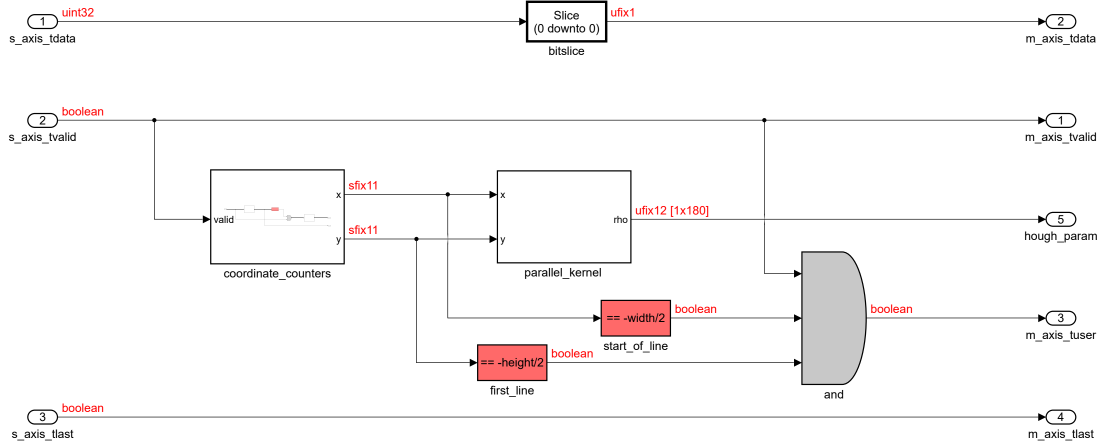


Figure B.5: The contents of the *parallel_lht_kernel* Simulink subsystem.

The purpose of the *coordinate_counters* block is to determine the coordinates of the current pixel, which can be used by the *parallel_kernel* block to compute (3.2). The *coordinate_counters* subsystem is presented in Figure B.6. There are two counters that compute the image coordinates and are configured with a step size of one. The range of the *x_counter* is $[-960, 959]$ and the range of the *y_counter* is $[-540, 539]$. Both counters are count-limited and will return to their starting value after reaching their upper limit. The relational operator in the diagram asserts when the output of the *x_counter* is equal to 959, which causes the *y_counter* to increment by one.

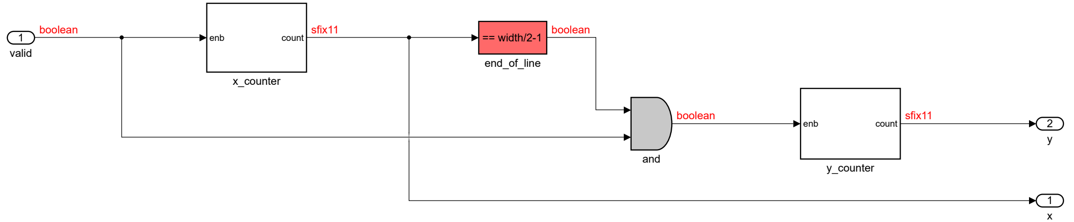


Figure B.6: The Simulink system for the *coordinate_counters* block.

Figure B.7 presents the contents of the *parallel_kernel* subsystem. Notice that there are two subsystems named *look_ahead_hough* and *rho_computation*. The add block is required to change the range of $\rho(\theta)$ from $[-D/2, D/2]$ to $[0, D]$. It is necessary to change the range so that it can be used to index the accumulator memory, which only accepts address values that are positive integers.

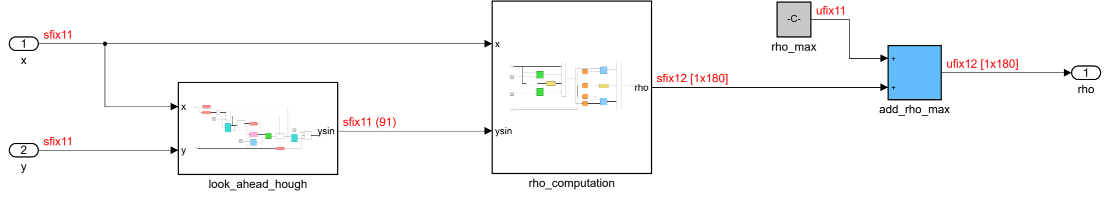


Figure B.7: The *parallel_kernel* Simulink subsystem.

The *look_ahead_hough* block implements the architecture design illustrated previously in Figure 3.19 on page 75. This architecture is also described extensively in [20]. Figure B.8 presents the Simulink model of the Look-Ahead Kernel. Various *HDL Coder* blocks are used in this design, which pre-computes $y\sin(\theta)$ for the next image row using a single multiplier.

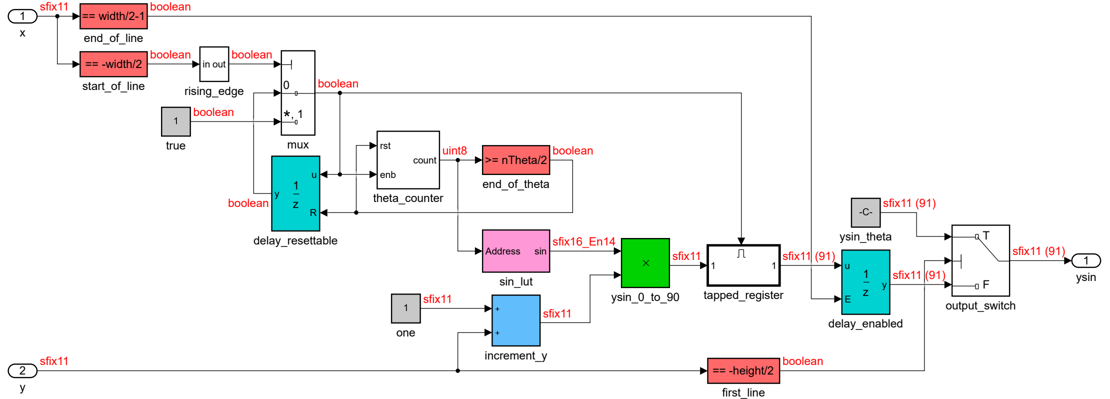


Figure B.8: Simulink model of the Look-Ahead Kernel, which is detailed in [20]. The design uses a single multiplier to pre-compute $y\sin(\theta)$ for the next image row. A tapped-register is employed to write values to an output register for use in the *rho_computation* subsystem.

Lastly, Figure B.9 presents the Simulink diagram for the *rho_computation* block, which computes (3.2). This parallel LHT kernel uses the resource-efficient technique presented in [87]. Many of the multiplications and additions are achieved using the vector processing capabilities of *HDL Coder* and Simulink. Notably, there is one additional optimisation implemented in this design that was not published in [87]. The multiplication of x and $\cos(60°)$ is equal to $x/2$, which can be implemented using a simple bit-shift to the right. This simple optimisation reduces the total DSP48E2 slice requirements by one.

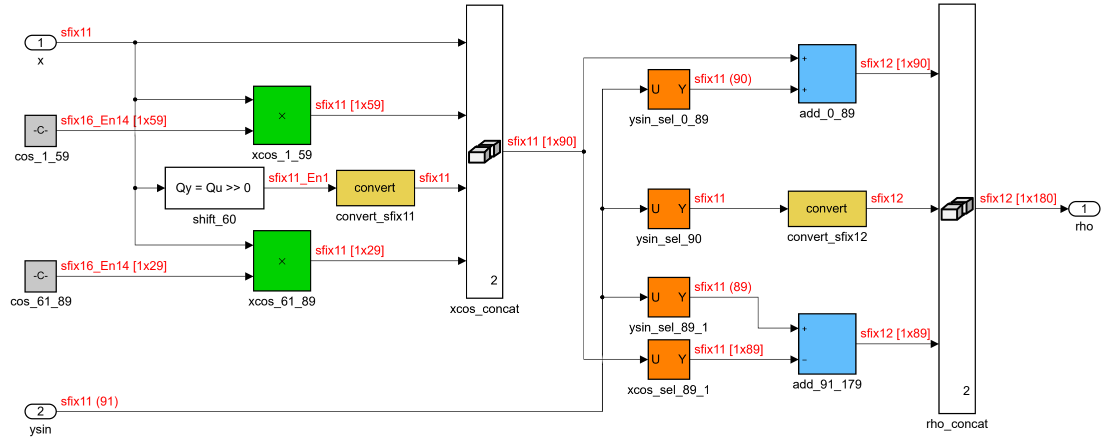


Figure B.9: The contents of the *rho_computation* subsystem, which implements (3.2) using the technique described in [87].

B.4 Parallel LHT Accumulator

Figure B.10 presents the Simulink model of the *parallel_lht_accumulator* subsystem, which contains a block called *accumulator_controller_fsm* and a subsystem named *accumulator_array*. There are relational operators, logical blocks and a switch that control the operation of the accumulator array.

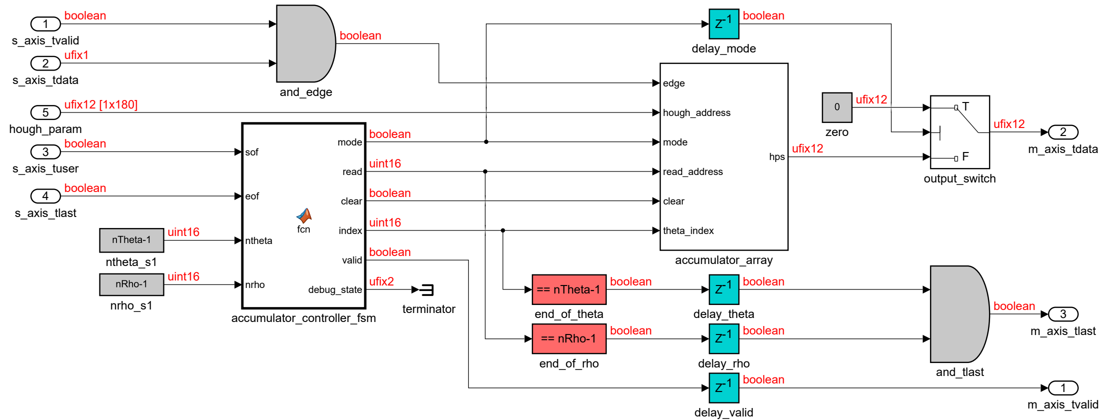


Figure B.10: The Simulink diagram of the *parallel_lht_accumulator* subsystem.

The *accumulator_controller_fsm* block implements a simple state machine to control the *accumulator_array* subsystem. The FSM was designed using the MATLAB programming language, as shown in Listing B.1. See the code comments in the listing for further information on the operation of the FSM.

Listing B.1: *accumulator_controller_fsm.m*

```
function [mode, read, clear, index, valid, debug_state] ...
    = fsm(sof, eof, ntheta, nrho)
% FSM Controls the voting of the parallel.accumulator array

% Initialise states as fi objects
IDLE = fi(0, 0, 2, 0);
VOTE = fi(1, 0, 2, 0);
READ = fi(2, 0, 2, 0);
CLEAR = fi(3, 0, 2, 0);

% Declare persistent objects for current state and counters
persistent current_state;
persistent rho_counter;
persistent theta_counter;

% Initialise persistent object current_state
if isempty(current_state)
    current_state = IDLE;
end

% Initialise persistent object rho_counter
if isempty(rho_counter)
    rho_counter = fi(0, 0, 16, 0);
end

% Initialise persistent object theta_counter
if isempty(theta_counter)
    theta_counter = fi(0, 0, 16, 0);
end

% Perform state transition and output assignment
switch current_state
    case IDLE
        % When IDLE, reset counters and no valid output
        rho_counter = fi(0, 0, 16, 0);
        theta_counter = fi(0, 0, 16, 0);
        clear = false;
        valid = false;
        debug_state = current_state;
        read = rho_counter;
```

Listing B.1 (Cont.): *accumulator_controller_fsm.m*

```
41            index = theta_counter;
42
43            % Transition from IDLE to VOTE is start of frame (sof) detected
44            % else, stay in IDLE
45            if sof
46                mode = true;
47                current_state = VOTE;
48            else
49                mode = false;
50                current_state = IDLE;
51            end
52
53        case VOTE
54            % When VOTE, keep counters at zero, keep mode True
55            rho_counter = fi(0, 0, 16, 0);
56            theta_counter = fi(0, 0, 16, 0);
57            mode = true;
58            clear = false;
59            valid = false;
60            debug_state = current_state;
61            read = rho_counter;
62            index = theta_counter;
63
64            % If end of frame (eof) detected, go to the READ state
65            % else, stay in the VOTE state
66            if eof
67                current_state = READ;
68            else
69                current_state = VOTE;
70            end
71
72        case READ
73            % To read out the HPS, cycle through the theta and rho counters.
74            % mode set to False and valid is True
75            mode = false;
76            clear = false;
77            read = rho_counter;
78            index = theta_counter;
79            valid = true;
80            debug_state = current_state;
81
```

Listing B.1 (Cont.): *accumulator_controller_fsm.m*

```
82          % If theta_counter and rho_counter are at limits then transition
83          % to the CLEAR state
84          % else, stay in the READ state
85          if theta_counter >= ntheta && rho_counter >= nrho
86              rho_counter = fi(0, 0, 16, 0);
87              theta_counter = fi(0, 0, 16, 0);
88              current_state = CLEAR;
89          elseif rho_counter >= nrho
90              theta_counter = fi(theta_counter + 1, 0, 16, 0);
91              rho_counter = fi(0, 0, 16, 0);
92              current_state = READ;
93          else
94              rho_counter = fi(rho_counter + 1, 0, 16, 0);
95              current_state = READ;
96          end
97
98      case CLEAR
99          % Clear output is set to True in this state to clear the memory
100         mode = false;
101         clear = true;
102         valid = false;
103         read = rho_counter;
104         index = theta_counter;
105         debug_state = current_state;
106
107         % The rho_counter and theta_counter are reset after full clear
108         % Then transition to the IDLE state
109         if rho_counter >= nrho
110             rho_counter = fi(0, 0, 16, 0);
111             theta_counter = fi(0, 0, 16, 0);
112             current_state = IDLE;
113         else
114             rho_counter = fi(rho_counter + 1, 0, 16, 0);
115             theta_counter = fi(0, 0, 16, 0);
116             current_state = CLEAR;
117         end
118
119     otherwise
120         rho_counter = fi(0, 0, 16, 0);
121         theta_counter = fi(0, 0, 16, 0);
122         mode = false;
```

Listing B.1 (Cont.): *accumulator_controller_fsm.m*

```
123             clear = false;
124             valid = false;
125             debug_state = current_state;
126             read = rho_counter;
127             index = theta_counter;
128             current_state = IDLE;
129     end
```

Figure B.11 presents the contents of the *accumulator_array* subsystem, which contains two switches, a selector block, and a subsystem named *angle_accumulator*. The parallel LHT accumulator contains two modes of operation. One mode is used to accumulate and write votes to memory, while the second mode allows the memory to be read and cleared. The *accumulator_array* subsystem contains two switches that allow the mode of operation to be easily changed during runtime. Also, the output of the *angle_accumulator* subsystem is a selector block, which facilitates angle indexing when reading the HPS from memory.

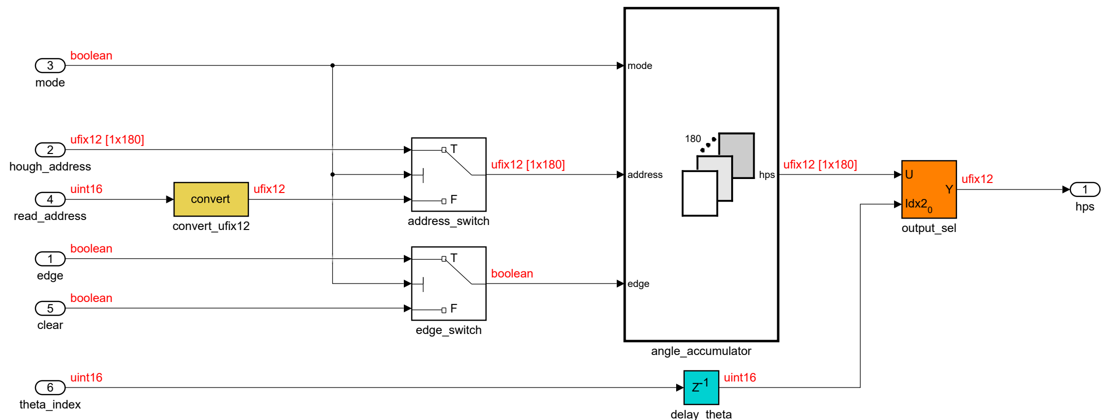


Figure B.11: The *accumulator_array* Simulink design.

The symbol inside the *angle_accumulator* subsystem above is used to represent a subsystem that has independent instances for each element of an input vector. In this case, the *angle_accumulator* subsystem will have 180 instances, as the input address signal is a vector that contains 180 elements. The contents of the *angle_accumulator* subsystem is presented in Figure B.12. Notice that there is a block named *For Each* at the top-left corner of the Simulink system, which enables this functionality.

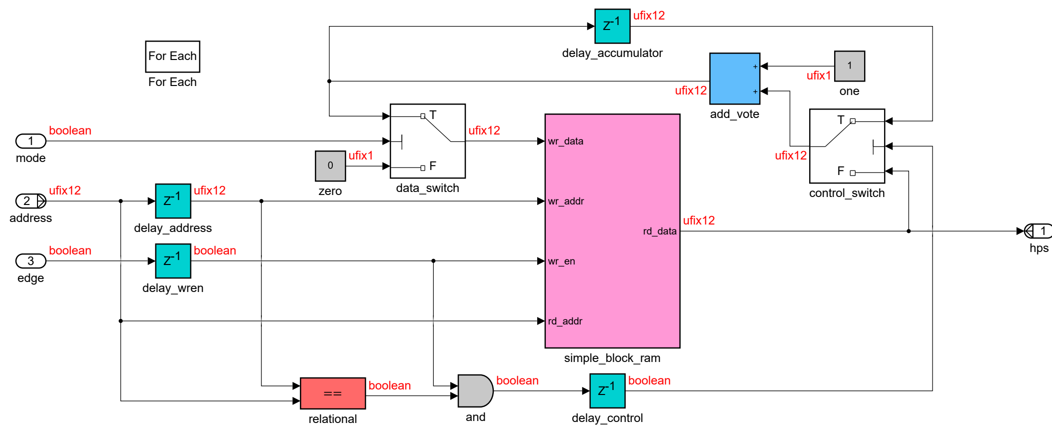


Figure B.12: The contents of the *angle_accumulator* Simulink subsystem.

The *angle_accumulator* subsystem contains a *simple_block_ram* block, which implements the accumulator memory for one angle in θ using BRAMs. The specific architecture design of this accumulator is detailed further in [14].

B.5 Standard LHT Software Model

The parallel LHT architecture was simulated and compared to a software model of the standard LHT in MATLAB. The source code for the software model of the standard LHT is presented in Listing B.2.

Listing B.2: *LineHoughTransform.m*

```
1  function [hps] = LineHoughTransform(edge)
2  %LineHoughTransform Applies the Standard Line Hough Transform to binary
3  %   images. The Standard Line Hough Transform (LHT) as defined by Duda &
4  %   Hart [1], is applied to binary edge images using this function. The
5  %   Hough Parameter Space (HPS) is produced by applying the analytical
6  %   function for lines as described in [1]. The HPS is returned as a
7  %   double precision array.
8  %
9  %   [1] - R. O. Duda, P. E. Hart, "Use of the Hough transformation to
10 %           detect lines and curves in pictures", Commun. ACM, vol. 15,
11 %           no. 1, pp. 11-15, Jan. 1972.
12 %
13
```

Listing B.2 (Cont.): *LineHoughTransform.m*

```
14  %Check arguments (no dRho as hardware model is dRho=1 for rounding)
15  switch nargin
16      case 0
17          error('Not enough input arguments.');
18      case 1
19          % Do Nothing
20      otherwise
21          error('Error occurred.');
22  end
23
24  %Get the dimensions of the input image.
25  [height, width] = size(edge);
26
27  %Get the maximum Rho
28  maxRho = ceil(sqrt((height/2)^2+(width/2)^2));
29
30  %Get theta
31  theta = 1:1:180;
32
33  % Initialise quantised cos & sin
34  cosQ = double(fi(cos(deg2rad(theta-1)), 1, 16, 14));
35  sinQ = double(fi(sin(deg2rad(theta-1)), 1, 16, 14));
36
37  %Get HPS dimensions
38  [~, m] = size(theta);
39  n = maxRho*2;
40
41  %Initialise the Accumulator array (A)
42  A = zeros(n, m);
43
44  %Get HPS by iterating through the binary edge image and operating on
45  %feature points.
46  for y = 1:height
47      for x = 1:width
48
49          % Iterate over binary edge pixels
50          if edge(y,x) > 0
51
52              % Adjust Cartesian Positions
53              xTemp = x-width/2-1;
54              yTemp = y-height/2-1;
```

Listing B.2 (Cont.): *LineHoughTransform.m*

```
55
56                 % Get Hough Parameters
57                 rho = round(xTemp*cosQ) + round(yTemp*sinQ) + maxRho + 1;
58
59                 % Apply votes to HPS
60                 for i = 1:m
61                     A(rho(1,i), theta(1,i)) = A(rho(1,i), theta(1,i)) + 1;
62                 end
63             end
64         end
65     end
66
67     %Assign the accumulator array to the HPS output array
68     hps = A;
69
70     end
```

B.6 Summary

This appendix initially introduced MathWorks *HDL Coder* blocks and fixed-point data types. The Simulink design of a parallel LHT architecture was then explored for implementation on an FPGA device. Several components of the LHT architecture were presented including the coordinate counters, Look Ahead Kernel, parallel LHT kernel, accumulator controller, and parallel accumulator memory. Lastly, a standard LHT software model in MATLAB code was presented.

Appendix C

Symmetric LHT Validation Results

The Symmetric LHT architecture was validated by the HEP using the Jupyter environment as shown in Figure C.1 and Figure C.2. The resulting HPS for each image was the same as that produced by the Symmetric LHT simulation and MATLAB software model.

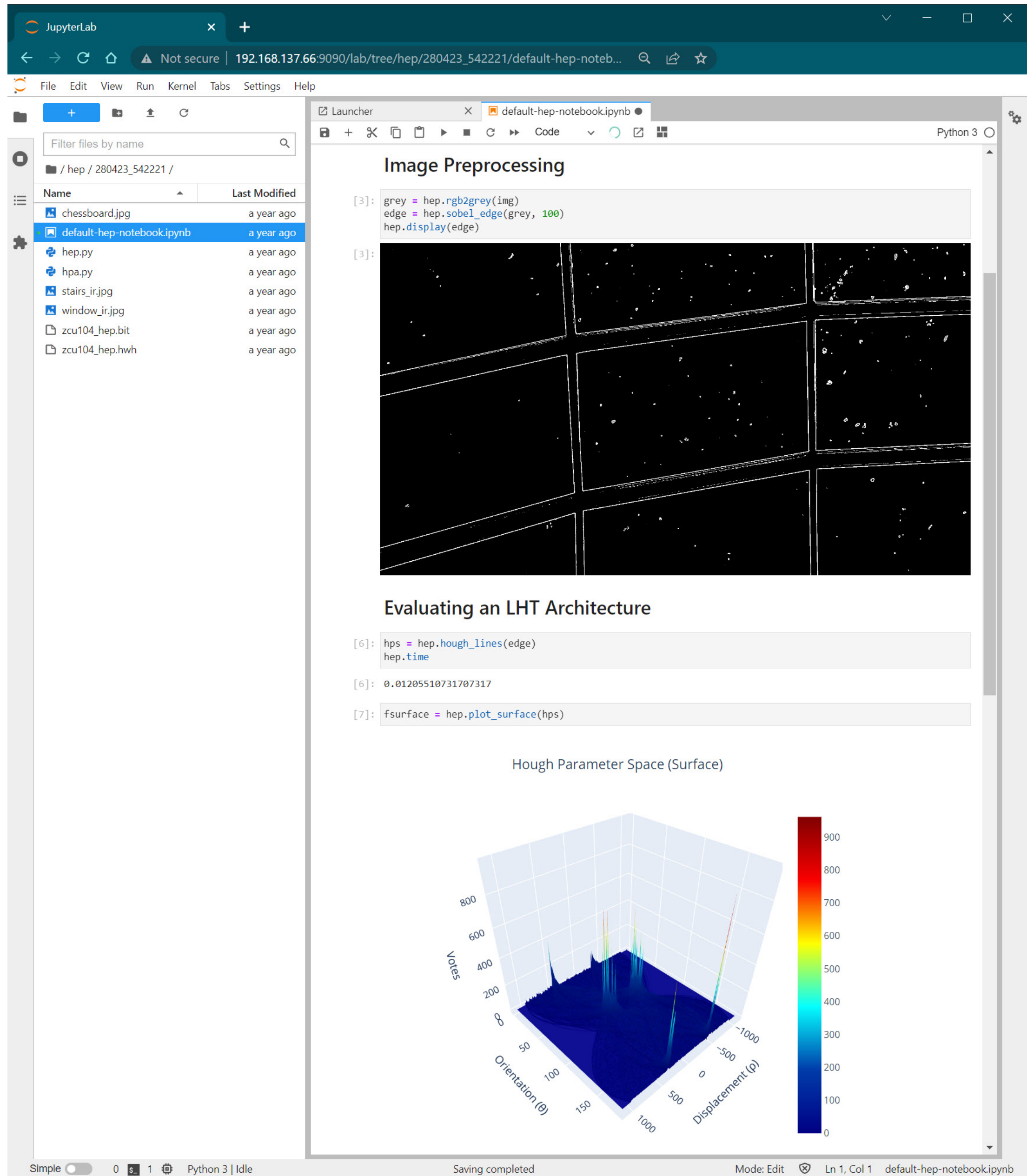


Figure C.1: This screenshot presents the HEP Jupyter environment, where the Symmetric LHT architecture is undergoing hardware validation using the window image.

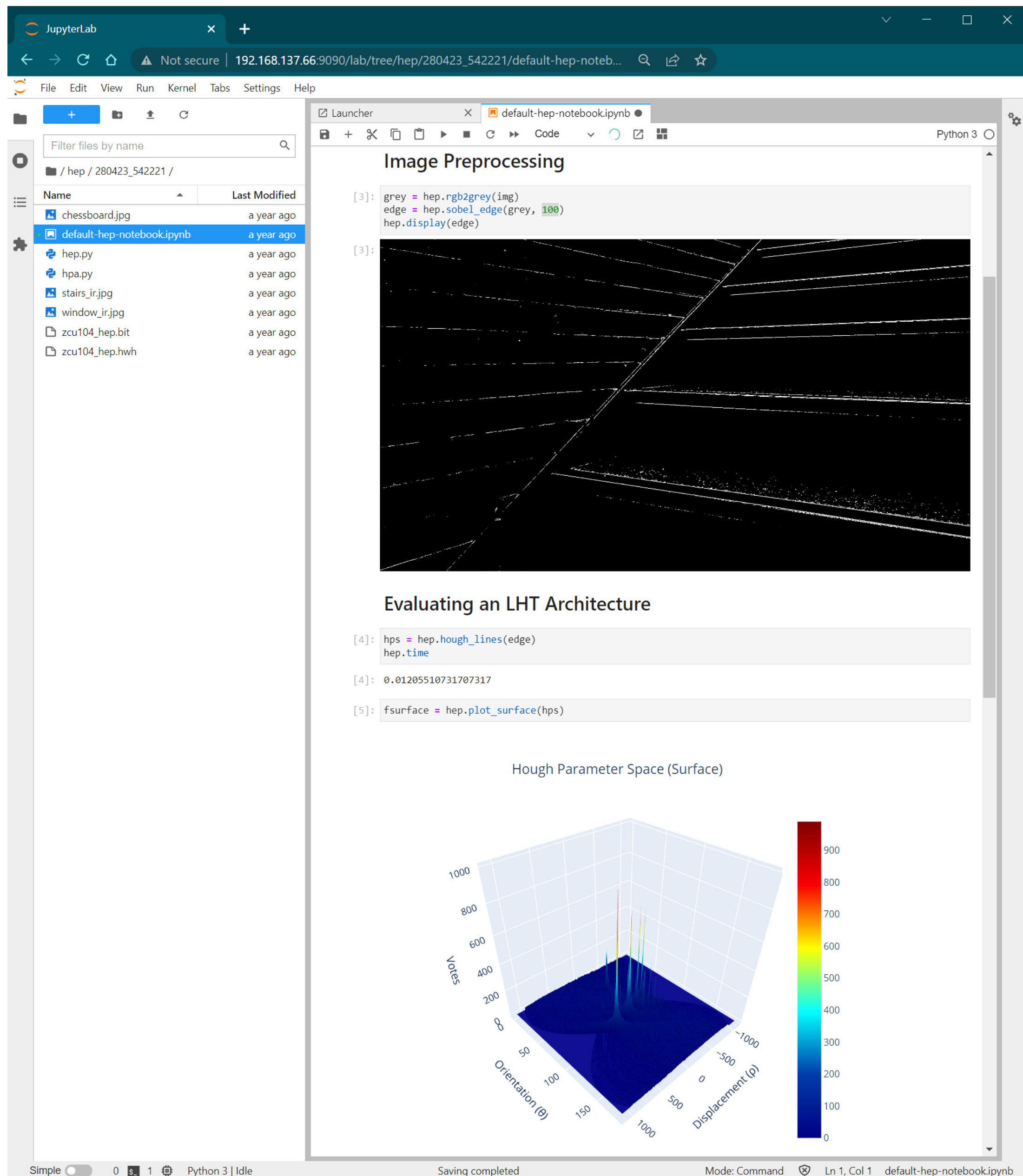


Figure C.2: This screenshot presents the HEP Jupyter environment, where the Symmetric LHT architecture is undergoing hardware validation using the stairs image.

Appendix D

ARLHT Validation Results

The ARLHT architecture was validated by the HEP using the Jupyter environment as shown in Figure D.1 and Figure D.2. The resulting HPS for each image was the same as that produced by the ARLHT simulation and MATLAB software model.

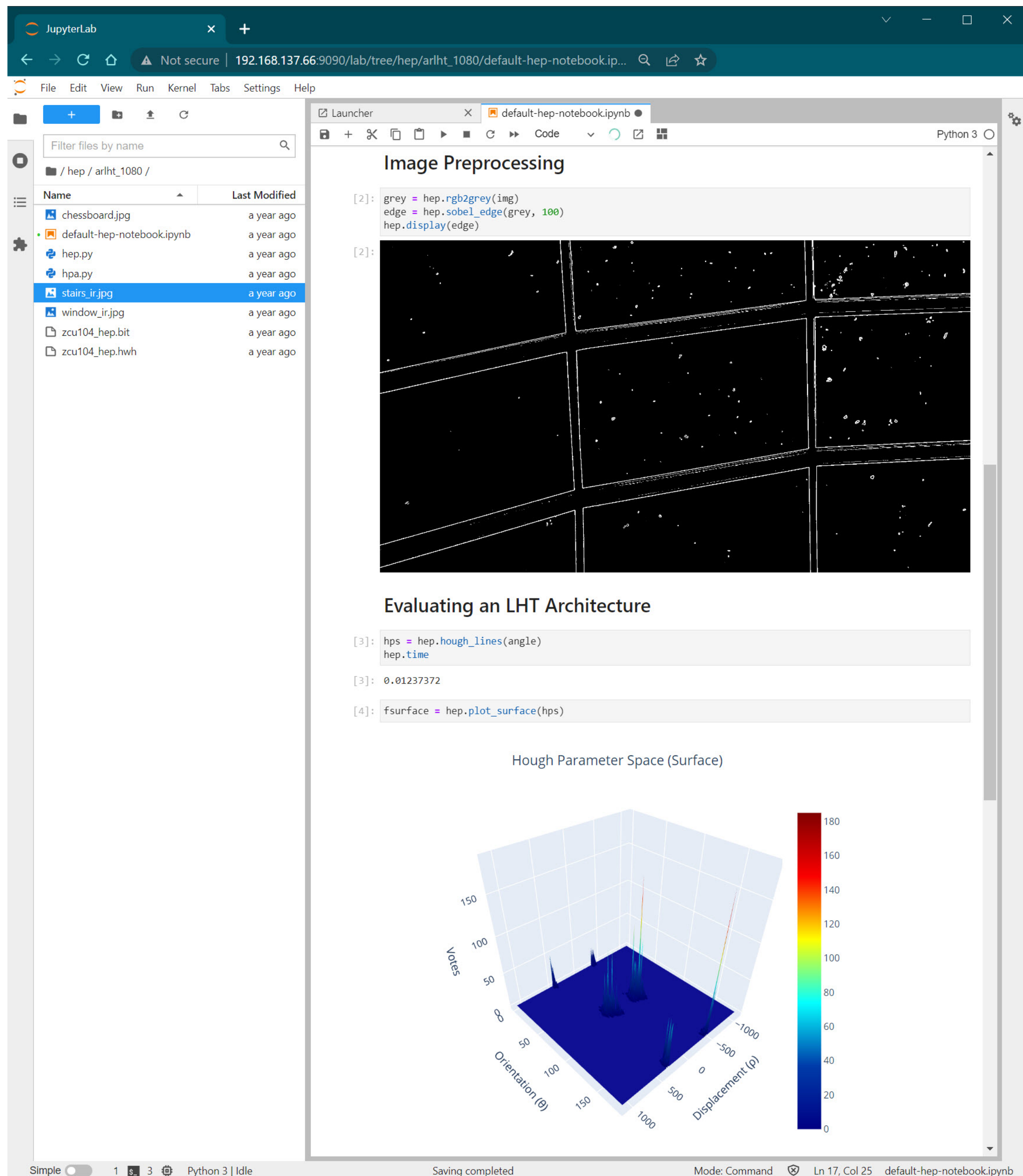


Figure D.1: This screenshot presents the HEP Jupyter environment, where the ARLHT architecture is undergoing hardware validation using the window image.

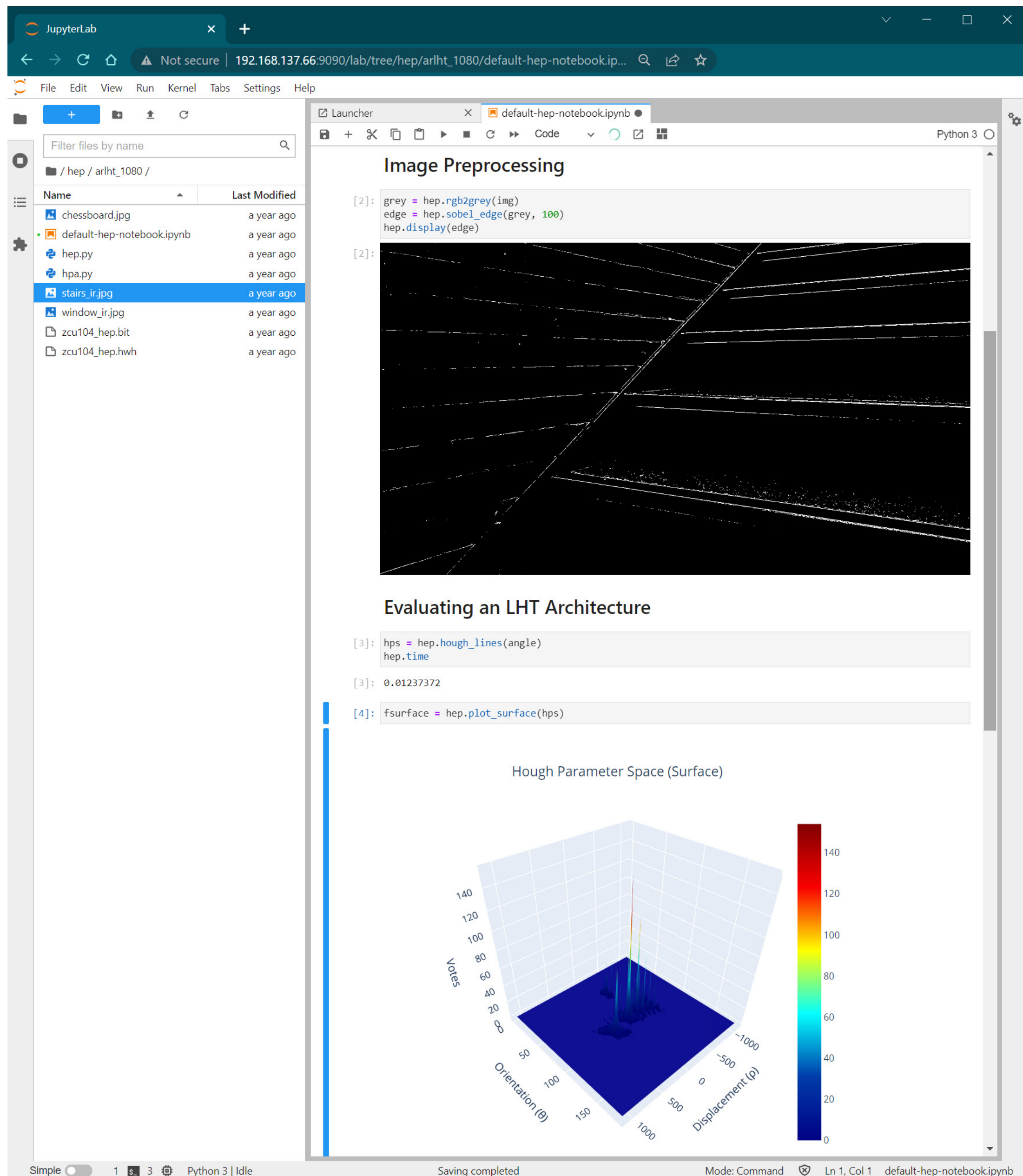


Figure D.2: This screenshot presents the HEP Jupyter environment, where the ARLHT architecture is undergoing hardware validation using the stairs image.

References

[1] L. Zhou, L. Zhang, and N. Konz, "Computer Vision Techniques in Manufacturing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 53, no. 1, pp. 105–117, 2023.

[2] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. Nielsen, "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms," *Signal Processing: Image Communication*, vol. 68, pp. 101–119, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0923596518303606

[3] R. O. Duda and P. E. Hart, "Use of the Hough transform to detect lines and cures in pictures," *Communications of the Association Computing Machinery*, vol. 15, no. 1, pp. 11–15, 1972.

[4] A. Kadyrov and M. Petrou, "The Trace transform and its applications," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 8, pp. 811–828, 2001.

[5] F. Chao, S. Yu-Pei, and J. Ya-Jie, "Multi-Lane Detection Based on Deep Convolutional Neural Network," *IEEE Access*, vol. 7, pp. 150 833–150 841, 2019.

[6] M. H. Nasseri, H. Moradi, S. Nasiri, and R. Hosseini, "Power Line Detection and Tracking Using Hough Transform and Particle Filter," in *2018 6th RSI International Conference on Robotics and Mechatronics (IcRoM)*, 2018, pp. 130–134.

[7] A. I. Purica, B. Pesquet-Popescu, and F. Dufaux, "A railroad detection algorithm for infrastructure surveillance using enduring airborne systems," in *2017 IEEE*

International Conference on Acoustics, Speech and Signal Processing (ICASSP), March 2017, pp. 2187–2191.

[8] S. E.-D. N. Mohamed, R. M. Al-Makhlasawy, M. Abdelaziz, A. A. M. Khalaf, M. I. Dessouky, and F. E. A. El-Samie, “Efficient utilization of Hough transform and orthogonal-triangular decomposition for optical wireless modulation format recognition,” *Appl. Opt.*, vol. 61, no. 4, pp. 875–883, Feb 2022. [Online]. Available: http://opg.optica.org/ao/abstract.cfm?URI=ao-61-4-875

[9] K. K. Guner, T. O. Gulum, and B. Erkmen, “FPGA-Based Wigner–Hough Transform System for Detection and Parameter Extraction of LPI Radar LFMCW Signals,” *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–15, 2021.

[10] F. O’Gorman and M. B. Clowes, “Finding Picture Edges Through Collinearity of Feature Points.” *IEEE Transactions on Computers*, vol. C-25, no. 4, pp. 449–456, 1976.

[11] L. A. F. Fernandes and M. M. Oliveira, “Real-time line detection through an improved Hough transform voting scheme,” *Pattern Recognition*, vol. 41, no. 1, pp. 299–314, 2008.

[12] J. Illingworth and J. Kittler, “The Adaptive Hough Transform,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, no. 5, pp. 690–698, 1987.

[13] X. Lu, L. Song, S. Shen, K. He, S. Yu, and N. Ling, “Parallel Hough Transform-based straight line detection and its FPGA implementation in embedded vision.” *Sensors (Basel, Switzerland)*, vol. 13, no. 7, pp. 9223–9247, 2013.

[14] X. Zhou, N. Tomagou, Y. Ito, and K. Nakano, “Efficient hough transform on the FPGA using DSP slices and block RAMs,” *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, pp. 771–778, 2013.

[15] D. G. Bailey, "Image Border Management for FPGA Based Filters," in *2011 Sixth IEEE International Symposium on Electronic Design, Test and Application*, Jan 2011, pp. 144–149.

[16] AMD, Inc., "AMD Homepage (Webpage)," July 2023. [Online]. Available: https://www.amd.com/ (accessed Jul. 25, 2023)

[17] Intel, Inc., "Intel Homepage (Webpage)," July 2023. [Online]. Available: https://www.intel.com/ (accessed Jul. 25, 2023)

[18] AMD, Inc., "Cost-Optimized Portfolio Product Selection Guide (XMP100)," July 2023. [Online]. Available: https://docs.xilinx.com/v/u/en-US/cost-optimized-product-selection-guide (accessed Jul. 25, 2023)

[19] Xilinx, Inc., "UltraScale+ FPGAs Product Tables and Product Selection Guide," July 2023. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf (accessed Jul. 25, 2023)

[20] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of the gradient-based Hough transform using DSP slices and block RAMs on the FPGA," *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, pp. 762–770, 2014.

[21] D. G. Bailey, "Streamed Hough Transform and Line Reconstruction on FPGA," *International Conference Image and Vision Computing New Zealand*, vol. 2017-Decem, pp. 1–6, 2018.

[22] I. El Hajjouji, S. Mars, Z. Asrih, and A. El Mourabit, "A novel FPGA implementation of Hough Transform for straight lane detection," *Engineering Science and Technology, an International Journal*, vol. 23, no. 2, pp. 274–280, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2215098618314782

[23] P.-K. Ser and W.-C. Siu, "Memory compression for straight line recognition using the Hough transform," *Pattern Recognition Letters*, vol. 16, no. 2, pp. 133 – 145, 1995.

[24] Xilinx Inc., "Zynq UltraScale+ MPSoC Product Selection Guide," April 2022. [Online]. Available: https://docs.xilinx.com/v/u/en-US/zynq-ultrascale-plus-product-selection-guide (accessed Apr. 13, 2023)

[25] ——, "PYNQ - Python Productivity for Zynq." [Online]. Available: http://www.pynq.io/ (accessed Apr. 13, 2023)

[26] The MathWorks Inc., "Mathworks HDL Coder Product Page," Natick, Massachusetts, United States, April 2022. [Online]. Available: https://uk.mathworks.com/products/hdl-coder.html (accessed Apr. 13, 2023)

[27] D. Northcote, L. H. Crockett, P. Murray, and R. W. Stewart, "A PYNQ Evaluation Platform for FPGA Architectures of the Line Hough Transform," in *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2020, pp. 133–137.

[28] D. Northcote, L. H. Crockett, and P. Murray, "FPGA Implementation of a Memory-Efficient Hough Parameter Space for the Detection of Lines," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[29] J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. H. Crockett, and R. W. Stewart, "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework," *IEEE Access*, vol. 8, pp. 129 012–129 031, 2020.

[30] L. H. Crockett, D. Northcote, C. Ramsay, F. D. Robinson, and R. W. Stewert, *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*, 2019.

[31] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and D. Northcote, *The Zynq Book Tutorials for Zybo and Zedboard*, 2015.

[32] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[33] D. DeTone, T. Malisiewicz, and A. Rabinovich, "SuperPoint: Self-Supervised Interest Point Detection and Description," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2018.

[34] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[35] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[36] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 77–84.

[37] onsemi, "VITA 2000 2.3 Megapixel 92 FPS Global Shutter CMOS Image Sensor," December 2016. [Online]. Available: https://www.onsemi.com/download/data-sheet/pdf/noiv1sn2000a-d.pdf (accessed Apr. 13, 2023)

[38] I. Lankshear, "The Economics of ASICs: At What Point Does a Custom SoC Become Viable?" July 2019. [Online]. Available: https://www.electronicdesign.com/technologies/embedded/article/21808278/ensilica-the-economics-of-asics-at-what-point-does-a-custom-soc-become-viable (accessed Jul. 24, 2023)

[39] Xilinx Inc., "UltraScale Architecture and Product Data Sheet: Overview, DS890 (v4.1.1)," February 2022. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds890-ultrascale-overview.pdf (accessed Apr. 13, 2023)

[40] ——, “UltraScale Architecture Configurable Logic Block: User Guide, UG574, (v1.5),” February 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb (accessed Apr. 13, 2023)

[41] S. White, “Applications of distributed arithmetic to digital signal processing: a tutorial review,” *IEEE ASSP Magazine*, vol. 6, no. 3, pp. 4–19, 1989.

[42] Xilinx Inc., “UltraScale Architecture DSP Slice: User Guide, UG579, (v1.11),” August 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp (accessed Apr. 13, 2023)

[43] ——, “FIR Compiler: LogiCORE IP Product Guide, PG149, (v7.2),” January 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg149-fir-compiler (accessed Apr. 13, 2023)

[44] ——, “Fast Fourier Transform: LogiCORE IP Product Guide, PG149, (v7.2),” August 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg109-xfft (accessed Apr. 13, 2023)

[45] ——, “UltraScale Architecture Memory Resources: User Guide, UG573, (v1.13),” September 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug573-ultrascale-memory-resources (accessed Apr. 13, 2023)

[46] ——, “UltraScale Architecture Libraries Guide,” December 2020. [Online]. Available: https://docs.xilinx.com/r/2020.2-English/ug974-vivado-ultrascale-libraries/Introduction (accessed Jun. 23, 2023)

[47] ——, “Zynq UltraScale+ Device: Technical Reference Manual, UG1085, (v2.2),” December 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1085-zynq-ultrascale-trm (accessed Apr. 13, 2023)

[48] ——, “Zynq-7000 SoC Technical Reference Manual, UG585, (v1.13),” April 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM (accessed Apr. 13, 2023)

[49] Arm Ltd., "Cortex-A53 MPCore Processor Technical Reference Manual, Issue G, Revision r0p4," February 2016. [Online]. Available: https://developer.arm.com/documentation/ddi0500/j/ (accessed Apr. 13, 2023)

[50] ——, "Cortex-R5 Technical Reference Manual, Issue D, Revision r1p2," September 2011. [Online]. Available: https://developer.arm.com/documentation/ddi0460/c (accessed Apr. 13, 2023)

[51] ——, "Mali GPU Developer Tools Technical Overview, Issue A, Version 1.0," October 2009. [Online]. Available: https://developer.arm.com/documentation/dui0501/a/ (accessed Apr. 13, 2023)

[52] Xilinx Inc., "H.264/H.265 Video Codec Unit, v1.2," July 2021. [Online]. Available: https://docs.xilinx.com/r/en-US/pg252-vcu/H.264/H.265-Video-Codec-Unit-v1.2 (accessed Apr. 13, 2023)

[53] Arm Ltd., "AMBA 4 Overview," April 2022. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba/amba-4 (accessed Apr. 13, 2023)

[54] ——, "AMBA AXI and ACE Protocol Specification, Issue E," February 2013. [Online]. Available: https://developer.arm.com/documentation/ihi0022/ (accessed Apr. 13, 2023)

[55] ——, "AMBA 4 AXI4-Stream Protocol Specification, Issue A, Version 1.0," March 2010. [Online]. Available: https://developer.arm.com/documentation/ihi0051 (accessed Apr. 13, 2023)

[56] Xilinx Inc., "Xilinx Vivado Design Suite Product Page," April 2022. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html (accessed Apr. 13, 2023)

[57] The MathWorks Inc., "Mathwork Simulink Product Page," Natick, Massachusetts, United States, April 2022. [Online]. Available: https://uk.mathworks.com/products/simulink.html (accessed Apr. 13, 2023)

[58] Xilinx Inc., "AXI DMA LogiCore IP Product Guide," June 2019. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg021_axi_dma (accessed Apr. 13, 2023)

[59] ——, "Model-Based DSP Design Using System Generator," June 2020. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_1/ug948-vivado-sysgen-tutorial.pdf (accessed Apr. 13, 2023)

[60] ——, "Vivado Design Suite User Guide: High-Level Synthesis, UG902, (v2020.1)," May 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis (accessed Apr. 13, 2023)

[61] P. S. Foundation, "Python 3.8.13 Documentation," April 2022. [Online]. Available: https://docs.python.org/3.8/ (accessed Apr. 13, 2023)

[62] P. Jupyter, "The Jupyter Project Webpage," April 2022. [Online]. Available: https://jupyter.org/ (accessed Apr. 13, 2023)

[63] Xilinx Inc., "Zynq UltraScale+ RFSoC Product Selection Guide," April 2022. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/zynq-usp-rfsoc-product-selection-guide.pdf (accessed Apr. 13, 2023)

[64] Xilinx, Inc., "ZCU104 Evaluation Board," 2023. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/zcu104.html (accessed Oct. 9, 2023)

[65] Xilinx Inc., "MicroBlaze Processor Reference Guide, UG984, (v2020.1)," June 2020. [Online]. Available: https://docs.xilinx.com/v/u/2020.1-English/ug984-vivado-microblaze-ref (accessed Apr. 13, 2023)

[66] R. C. Gonzalez and R. E. Woods, *Digital image processing.* Upper Saddle River, N.J.: Prentice Hall, 2008. [Online]. Available: http://www.amazon.com/Digital-Image-Processing-3rd-Edition/dp/013168728X

[67] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*, 2011.

[68] J. M. S. Prewitt, *"Object Enhancement and Extraction" Picture Processing and Psychopictorics*, 1970.

[69] I. Sobel, "An isotropic 3 by 3 image gradient operator," *Machine Vision for three-demensional Sciences*, vol. 1, no. 1, pp. 23–34, 1990.

[70] J. Canny, "A computational approach to edge detection." *IEEE transactions on pattern analysis and machine intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[71] I. Abdou and W. Pratt, "Quantitative design and evaluation of enhancement/thresholding edge detectors," *Proceedings of the IEEE*, vol. 67, no. 5, pp. 753–763, 1979.

[72] J. E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sep. 1959.

[73] J. Walther, "A Unified Algorithm for Elementary Functions," *Conference Proceedings, Spring Joint Computer Conference*, pp. 379–385, May 1971.

[74] R. Andraka, "A Survey of CORDIC Algorithms for FPGA Based Computers," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 191–200. [Online]. Available: https://doi.org/10.1145/275107.275139

[75] P. V. C. Hough, "Machine analysis of bubble chamber pictures," *2nd International Conference on High-Energy Accelerators and Instrumentation*, vol. 73, pp. 554–558, 1959.

[76] D. Ballard, "Generalizing the Hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, no. 2, pp. 111 – 122, 1981. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0031320381900091

[77] A. Rosenfeld, *Picture Processing by Computer*, 1969.

[78] P. E. Hart, "How the Hough transform was invented [DSP History]," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 18–22, 2009.

[79] J. Immerkær, "Some remarks on the straight line Hough transform," *Pattern Recognition Letters*, vol. 19, no. 12, pp. 1133 – 1135, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865598000956

[80] H. Koshimizu and M. Numada, "FIHT2 AlGORITHM : A FAST INCREMENTAL HOUGH TRANSFORM," in *IAPR Workshop on Machine Vision Applications*, 1990, pp. 233–236.

[81] N. Kiryati and Y. Eldar, "A Probabilistic Hough Transform," *Pattern Recognition*, vol. 24, no. 4, pp. 303–316, 1991.

[82] J. Matas, C. Galambos, and J. Kittler, "Robust Detection of Lines Using the Progressive Probabilistic Hough Transform," *Computer Vision and Image Understanding*, vol. 78, no. 1, pp. 119–137, 2000.

[83] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

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

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

[86] P. Lee and A. Evagelos, "An implementation of a multiplierless Hough transform on an FPGA platform using hybrid-log arithmetic," *Real-Time Image Processing 2008*, vol. 6811, no. March, p. 68110G, 2008.

[87] X. Zhou, Y. Ito, and K. Nakano, "An Efficient Implementation of the Hough Transform Using DSP Slices and Block RAMs on the FPGA," in *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, 2013, pp. 85–90.

[88] A. Elhossini and M. Moussa, "A memory efficient FPGA implementation of hough transform for line and circle detection," *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering: Vision for a Greener Future, CCECE 2012*, pp. 1–5, 2012.

[89] Z. H. Chen, A. W. Su, and M. T. Sun, “Resource-efficient FPGA architecture and implementation of hough transform,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 8, pp. 1419–1428, 2012.

[90] P. Solod, N. Jindapetch, K. Sengchuai, A. Booranawong, P. Hoyingcharoen, S. Chumpol, and M. Ikura, “Memory Optimization for Accelerating Hough Transform on FPGA using High Level Synthesis,” in *2019 IEEE International Circuits and Systems Symposium (ICSyS)*, 2019, pp. 1–4.

[91] The MathWorks Inc., “Lane Departure Warning System,” Natick, Massachusetts, United States, 2023. [Online]. Available: https://uk.mathworks.com/help/vision/ug/lane-departure-warning-system-1.html (accessed Apr. 13 2023)

[92] M. Stokkeland, K. Klausen, and T. A. Johansen, “Autonomous visual navigation of Unmanned Aerial Vehicle for wind turbine inspection,” in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015, pp. 998–1007.

[93] H. Yu, W. Yang, H. Zhang, and W. He, “A UAV-based crack inspection system for concrete bridge monitoring,” in *2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2017, pp. 3305–3308.

[94] R. A. Arango Quiroz, F. Pereira Guidotti, and A. E. Bedoya, “A method for automatic identification of crop lines in drone images from a mango tree plantation using segmentation over YCrCb color space and Hough transform,” in *2019 XXII Symposium on Image, Signal Processing and Artificial Vision (STSIVA)*, 2019, pp. 1–5.

[95] M. D. Bah, A. Hafiane, and R. Canals, “CRowNet: Deep Network for Crop Row Detection in UAV Images,” *IEEE Access*, vol. 8, pp. 5189–5200, 2020.

[96] S. Nashat, A. Abdullah, and M. Abdullah, “Machine vision for crack inspection of biscuits featuring pyramid detection scheme,” *Journal of Food Engineering*, vol. 120, pp. 233 – 247, 2014.

[97] J. Wang, P. Fu, and R. X. Gao, "Machine vision intelligence for product defect inspection based on deep learning and Hough transform," *Journal of Manufacturing Systems*, vol. 51, pp. 52–60, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0278612519300111

[98] P. T. Inc., "Collaborative data science," Montreal, QC, 2023. [Online]. Available: https://plot.ly (accessed Apr. 13, 2023)

[99] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[100] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[101] Bokeh Development Team, *Bokeh: Python library for interactive visualization*, 2018. [Online]. Available: https://bokeh.pydata.org/en/latest/

[102] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. [Online]. Available: https://doi.org/10.21105/joss.03021

[103] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. [Online]. Available: https://ggplot2.tidyverse.org

[104] The MathWorks Inc., "plot: 2-D line plot," Natick, Massachusetts, United States, 2023. [Online]. Available: https://www.mathworks.com/help/matlab/ref/plot.html (accessed Apr. 13, 2023)

[105] ——, "Board and Reference Design System," Natick, Massachusetts, United States, 2023. [Online]. Available: https://uk.mathworks.com/help/hdlcoder/ug/board-and-reference-design-system.html (accessed Apr. 13, 2023)

[106] D. Northcote (Creator), L. H. Crockett, P. Murray, and R. W. Stewart, "The LHT Evaluation Platform," July 2020. [Online]. Available: https://doi.org/10.15129/4ec18371-3911-4e3e-823a-c65e38b7830f (accessed Apr. 13, 2023)

[107] S. Tatham, "PuTTY is an SSH and telnet client, developed originally by Simon Tatham for the Windows platform." 2023. [Online]. Available: https://www.putty.org/ (accessed Apr. 13, 2023)

[108] Xilinx Inc., "Doc PYNQ Libraries Interrupts," 2021. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/pynq_libraries/interrupt.html (accessed Apr. 13, 2023)

[109] ——, "Doc PYNQ Libraries," 2021. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/pynq_libraries.html (accessed Apr. 13, 2023)

[110] ——, "Doc PYNQ Libraries Overlay Source," 2021. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/_modules/pynq/overlay.html#Overlay (accessed Apr. 13, 2023)

[111] ——, "Doc PYNQ Libraries DMA Source," 2021. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/_modules/pynq/lib/dma.html#DMA (accessed Apr. 13, 2023)

[112] ——, "Doc PYNQ Libraries Default IP Source," 2021. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/_modules/pynq/overlay.html#DefaultIP (accessed Apr. 13, 2023)

[113] The MathWorks Inc., "HDL Coder Documentation," Natick, Massachusetts, United States, 2023. [Online]. Available: https://uk.mathworks.com/help/hdlcoder/ (accessed Apr. 13, 2023)

[114] ——, "FPGA-in-the-Loop Simulation," Natick, Massachusetts, United States, 2023. [Online]. Available: https://uk.mathworks.com/help/hdlverifier/ug/fpga-in-the-loop-fil-simulation.html (accessed Jul. 25, 2023)

[115] ——, "Resource Sharing," Natick, Massachusetts, United States, 2023. [Online]. Available: https://uk.mathworks.com/help/hdlcoder/ug/resource-sharing.html (accessed Jul. 22, 2023)

[116] Intel, "Cyclone V Device Handbook: Volume 1: Device Interfaces and Integration," July 2022. [Online]. Available: https://www.intel.com/programmable/technical-pdfs/683375.pdf (accessed Apr. 13, 2023)

[117] D. Electronics, "AMD XCZU7EV-2FFVC1156E." [Online]. Available: https://www.digikey.co.uk/en/products/detail/amd/XCZU7EV-2FFVC1156E/7034723 (accessed Jun. 13, 2023)

[118] Xilinx Inc., "7 Series Product Selection Guide," April 2021. [Online]. Available: https://docs.xilinx.com/v/u/en-US/7-series-product-selection-guide (accessed Jun. 13, 2023)

[119] D. Electronics, "AMD XC7A50T-1FTG256C." [Online]. Available: https://www.digikey.co.uk/en/products/detail/amd/XC7A50T-1FTG256C/5039080 (accessed Jun. 13, 2023)

[120] S. Du, Z. Dong, Y. Li, and T. Ikenaga, "Straight-Line Detection Within 1 Millisecond Per Frame for Ultrahigh-Speed Industrial Automation," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 4, pp. 5965–5975, 2023.

[121] The MathWorks Inc., "Fixed-Point Numbers in Simulink," Natick, Massachusetts, United States, 2023. [Online]. Available: https://uk.mathworks.com/help/fixedpoint/ug/fixed-point-numbers.html (accessed Apr. 13, 2023)