Local Descriptor Image Matching Acceleration and its Hardware Implementation

by

Parastoo Soleimani
B.Sc., University of Tehran, 2015
M.Sc., K. N. Toosi University of Technology, 2018

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Parastoo Soleimani, 2023
University of Victoria

All rights reserved. This dissertation may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Local Descriptor Image Matching Acceleration and its Hardware Implementation

by

Parastoo Soleimani
B.Sc., University of Tehran, 2015
M.Sc., K. N. Toosi University of Technology, 2018

Supervisory Committee

Dr. Kin Fun Li, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. David W. Capson, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Hausi A. Müller, Outside Member
(Department of Computer Science)

Supervisory Committee

Dr. Kin Fun Li, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. David W. Capson, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. Hausi A. Müller, Outside Member
(Department of Computer Science)

ABSTRACT

Computer vision algorithms have been used in an increasing number of applications during the past decades. One of the foremost challenges for using computer vision algorithms in practical applications is computational intensity which in turn may impact performance. In this dissertation, the focus is on improving speed performance by proposing novel algorithmic and hardware design techniques. Contributions are described for feature extraction and image matching.

Histogram of Oriented Gradients (HOG) is one of the commonly-used algorithms for feature extraction. In order to increase the speed of computation, a hardware-software co-design is presented. The proposed design makes four contributions, including a new task allocation method which reduces resource utilization, logarithm-based bin assignment which reduces latency, parallel histogram generation for latency reduction, and a simplified block normalization technique for reducing resource utilization. The proposed design of the HOG algorithm attains comparable frame rates and is shown to use fewer hardware resources in comparison with existing work in the literature.

Further contributions of this dissertation are related to the various steps of image matching algorithms, including scale-space generation, descriptor computation, and descriptor matching.

For scale-space generation, a real-time FPGA-based implementation of the AKAZE algorithm with non-linear scale-space generation is proposed. The proposed implementation makes two main contributions, that include (1) mapping the two passes of the AKAZE algorithm onto a hardware architecture for parallel processing of multiple image sections, and (2) designing multi-scale line buffers for reducing resource utilization. A frame rate of 304 frames per second for a 1280×768 image resolution is achieved which is shown to be faster in comparison with other published work.

For feature description, a novel circular shifting binary descriptor is proposed which leads to an efficient rotation invariant image matching. This new method eliminates complex operations such as multiplication and division from the orientation estimation step and thus significantly lowers the number of operations for descriptor computation.

For descriptor matching, a novel content-addressable memory (CAM) architecture is proposed which significantly accelerates the matching step of the image matching pipeline. The time complexity of the proposed modified CAM approach to binary descriptor matching is $O(n)$ while typically-used methods for matching have time complexity of $O(n^2)$. The resource utilization and timing metrics for several experiments are reported to demonstrate the efficacy of the proposed design.

Finally, the circular binary shifting descriptor and novel CAM matching design are applied to an experimental real-world application in aerial image matching to demonstrate the capabilities of the proposed methods.

Contents

Supervisory Committee ii

Abstract iii

Contents v

List of Tables viii

List of Figures ix

List of Acronyms xiii

Acknowledgements xiv

1 Introduction 1

1.1 Motivation 2

1.1.1 Image matching algorithm 2

1.1.2 Histogram of Oriented Gradients (HOG) 2

1.2 An Introduction to the HOG Algorithm and its Application 2

1.3 Image Matching Applications 3

1.4 An Introduction to Image Matching Algorithms 4

1.5 Content-addressable Memory for Key-point Matching 5

1.6 Hardware Acceleration of Image Processing Algorithms 6

1.7 Acknowledgment of Author's Contributions 7

1.8 Contributions and Dissertation Structure 7

2 Literature Review 10

2.1 Survey of Related Work on HOG Algorithm and its Hardware-Software Implementations 10

2.1.1 HOG Algorithm and SVM classifier 10

2.1.2 Hardware–Software Co-design Implementations 12

2.2 Survey of Related Work on Image Matching Algorithms 15

2.2.1 Scale-space Generation 15

2.2.2 Key-point Detection 16

2.2.3 Patch Description 17

2.2.4 Key-point Matching 21

2.3 Survey of Related Work on Hardware Implementations of Image Matching 21

2.3.1 Scale-space hardware implementation of the AKAZE algorithm 21
2.3.2 FPGA-based implementation of binary descriptor based image matching . . . 22
2.4 Survey of Related Work on CAM Architectures and Implementations 23
2.5 Conclusion 26

3 A Novel Hardware–Software Co-Design and Implementation of the HOG Algorithm 27
3.1 Hardware-Software Co-Design of the HOG-SVM System 27
3.2 HOG-SVM Core 28
3.2.1 Deserializer and Buffer Validity Check 29
3.2.2 Gradient and Magnitude Calculation 30
3.2.3 Logarithm-Based Bin Assignment 31
3.2.4 One-Row Histogram Generator 33
3.2.5 One-Cell Histogram Buffers 34
3.2.6 Two-Row Histogram Buffers 35
3.2.7 Block Normalization 35
3.2.8 SVM Classifier 37
3.3 Results and Comparison with Other Work 38
3.4 Conclusion 43

4 Real-time FPGA-based Implementation of the AKAZE Algorithm with Non-linear Scale-space Generation using Image Partitioning 44
4.1 A Brief Introduction to Accelerated KAZE (AKAZE) Non-linear Scale-space Generation 44
4.2 Hardware Implementation 46
4.2.1 Stage 1 - The Preprocessing Unit 47
4.2.2 Stage 2 - Diffusivity Calculation 49
4.2.3 Stage 3 - FED Filtering 49
4.2.4 Memory Management Unit 52
4.2.5 Image Resizer 53
4.3 Timing Analysis 55
4.4 Experimental Results 56
4.5 Conclusion 60

5 A Circular Shifting Binary Descriptor for Efficient Rotation Invariant Image Matching 61
5.1 Circular Shifting Binary Descriptor 61
5.2 Experiments 65
5.2.1 Number of Operations 66
5.2.2 Comparison of Rotation Error 66
5.2.3 Comparison of Image Matching Accuracy 67
5.3 Conclusion 68

6 A Modified CAM Architecture for Improving Binary Descriptor Matching 71

6.1 A New Architecture for Binary Descriptor Matching Based on CAM 72
6.1.1 A Novel Modified Partitioned CAM for Binary Descriptor Matching 72
6.1.2 Selection of the Number of Bits for Partitioning the Query Descriptor 80
6.1.3 Process Timing of the Proposed Modified Partitioned CAM 81
6.2 Experimental Results 83
6.2.1 Time Complexity 83
6.2.2 Speed Comparison 84
6.2.3 Accuracy of Image Matching 86
6.2.4 Hardware Implementation Metrics 89
6.3 Conclusion 91

7 Real-time Aerial Image Matching using Circular Shifting Binary Descriptors and CAM-based Matching 92
7.1 Experimental Setup 92
7.2 Experimental Results 94
7.3 Conclusion 97

8 Conclusion 99
8.1 Summary of Contributions 99
8.2 Future Work 101

List of Tables

Table 3.1 Limits for bin assignment 33
Table 3.2 Block normalization decoding method 36
Table 3.3 Comparison with other work 39
Table 3.4 HOG-SVM IP-core resources 41
Table 3.5 Resource usage of the whole hardware-software system 41

Table 4.1 Resource consumption of the stages of the algorithm 57
Table 4.2 Comparison of design metrics 57

Table 5.1 Comparison of the number of operations required for orientation compensation 66
Table 5.2 Comparison of descriptors in rotation error on HPatches dataset 67

Table 6.1 Comparison of binary descriptor matching speed of our design with other work 85
Table 6.2 Comparison of precision and recall metrics 88
Table 6.3 Resource utilization for various number of bits 89
Table 6.4 Resource utilization with various number of key-points 90
Table 6.5 Power usage and maximum operating frequency for various number of bits and key-points 90
Table 6.6 Resource utilization metrics for 500 key-points and 1920×1080 images on KCU105 91

List of Figures

Figure 1.1 Block diagram of feature-based image matching algorithms 4
Figure 1.2 Comparison of the functionality of (a) random access memory structure and (b) CAM structure. 6
Figure 1.3 The trade-offs in using various platforms for the implementation of computer vision algorithms. 7

Figure 2.1 A flowchart of the HOG algorithm from input image sensor to HOG features output. 11
Figure 2.2 Visualization of cells (4 by 4 pixels) and blocks (each containing 4 cells) in the HOG algorithm. 12
Figure 2.3 An example of linear scale-space generation with 3 octaves and 4 sub-scales in each octave. Images are from the Oxford affine covariant features dataset [34]. 16
Figure 2.4 An example of pixels in a patch around a key-point for FAST key-point detection computation. The center pixel is the key-point. The circle around the key-point for comparison are numbered for reference. 17
Figure 2.5 An example of orientation handling in conventional patch description algorithms for an image patch. The patch is an image from the Graffiti sequence in the Oxford affine covariant features dataset [34]. 19
Figure 2.6 An example of regions of SGLOH descriptor. The number of co-centered rings n_{rings} is 2, the number of sectors $n_{sectors}$ is 8, and the number of regions is 16. This figure is based on Fig. 1 in [51]. 20
Figure 2.7 (a) A general structure of CAM and (b) an example of finding query data using CAM structure. 24
Figure 2.8 (a) Basic implementation of CAM and (b) partitioned implementation of CAM. The total number of bits for the query input (N_b) is partitioned into m-bit address strings for k CAM units. The data stored in each CAM unit has N bits where N is the number of data locations in the data memory. . . 25

Figure 3.1 Block diagram of the proposed design and port connections 29
Figure 3.2 The overall diagram of the HOG-SVM core. 30
Figure 3.3 Line buffers in the deserializer module. 30
Figure 3.4 Difference between the slope of three logarithm functions. 32
Figure 3.5 The bin assignment procedure. 32

Figure 3.6 One-row histogram generation module. 34
Figure 3.7 The time-sharing protocol for the histogram generation module. 34
Figure 3.8 The block diagram of one-cell histogram buffers. 35
Figure 3.9 The block diagram of two-row histogram buffers. 35
Figure 3.10 The block diagram of block normalization module. 36
Figure 3.11 The SVM classifier module. 37
Figure 3.12 SVM block internal logic. 38
Figure 3.13 The relation between pixel stride and frame rate. 40
Figure 3.14 Comparison of software implementation and our proposed HW-SW co-design. 42

Figure 4.1 Pseudocode of AKAZE algorithm 45
Figure 4.2 Scharr filter weights 46
Figure 4.3 Block diagram of AKAZE scale-space generation with four channels 47
Figure 4.4 Data flow of the algorithm. The FED stage starts after the diffusivity stage. The preprocessing stage only processes the data once at the beginning of the algorithm while the diffusivity and FED stages run in each iteration 47
Figure 4.5 Preprocessing stage architecture. This stage contains two Gaussian filter modules. The output of the 9-row line buffer is a 9 by 9 window and the output of the 5-row line buffer is a 5 by 5 window. This stage computes the contrast factor and stores the filtered image in the memory 48
Figure 4.6 Block diagram of contrast factor calculation module 49
Figure 4.7 Diffusivity channel architecture 50
Figure 4.8 FED cell architecture 50
Figure 4.9 FED block architecture which contains two line buffers, an FED cell, and an adder 51
Figure 4.10 FED channel architecture which consists of 8 FED blocks 51
Figure 4.11 4 FED channels working in parallel 51
Figure 4.12 An example of the artifact from processing four sections of the image in parallel. Image from the Oxford affine covariant features dataset [34] 53
Figure 4.13 An example of selecting three phases for reading data from various sections of the memories. We show the data flow for diffusivity channel 2 as an example. In phase 1, this channel reads the data from the last two rows of the first section of the image. In phase 2, data enter channel 2 from the second section and in phase 3, diffusivity channel 2 reads the first two rows of data from the next section. Other channels have a similar data flow. Image from the Oxford affine covariant features dataset [34] 54
Figure 4.14 The architecture of the 3-row line buffer with multi-scale capability 55
Figure 4.15 Power consumption. The left diagram shows the portion of power consumed by different stages of the algorithm. The right diagram shows the dynamic and static power consumption. Total power consumption of the design is 1095mW 58

Figure 4.16 Comparison of repeatability between the software implementation and the hardware implementation based on simulation using image sets of the Oxford affine covariant features dataset [34] 59

Figure 5.1 An example of sample pairs in a seed. In this seed, $m = 6$ samples points and $n_{pairs} = 15$ pairs are shown. The sample point on the origin (0,0) is the key-point. 62

Figure 5.2 An example of sample points generated by rotating the sample seed $num_{rotations} = 16$ times. Each sample seed covers $\theta_{seed} = 22.5$ degrees. 63

Figure 5.3 An example of rotating a sector of the sampling pattern (top) and its effect on shifting the descriptors (bottom) in our proposed method. Symbols b_1 to b_{240} represent 240 bits of the descriptor. 64

Figure 5.4 Examples of sample seeds (top) and the complete sample pairs generated (bottom). The left images represents 3 pairs. The middle images represents 7 pairs and the right images represents 15 pairs. 64

Figure 5.5 An example of our proposed method for two patches from the same key-point with different orientations from the first and second images. 65

Figure 5.6 An example of circular rotation of the descriptor vector in our proposed method for two descriptor vectors from the same key-point with different orientations. 65

Figure 5.7 Comparison of rotation errors of our proposed method with ORB, BRISK, and SURF on the HPatches dataset [82]. 68

Figure 5.8 Examples comparing the mAP of our proposed method with the ORB algorithm. 69

Figure 5.9 Comparison of our method and the ORB descriptor over the HPatches dataset [82]. 70

Figure 6.1 An example of finding a match in a RAM using partitioned CAM units for conventional partitioned CAM and our approach for modified partitioned CAM. (a) Shows the query data and three CAM units, the selected content based on the input index is highlighted for each CAM unit. (b) Shows the outputs of the conventional partitioned CAM and our modified partitioned CAM. (c) Shows the content of the corresponding RAM. The selected content in the RAM unit is highlighted. 75

Figure 6.2 Block diagram of our modified partitioned CAM approach for binary descriptor matching. The inputs are a key-point of the target image with its corresponding descriptor. The outputs are a key-point in the target image and a key-point in the reference image that are matched based on their binary descriptors distance. 77

Figure 6.3 (a) The architecture of our matching module using partitioned CAM. The gray boxes are registers. The summation result is computed using N adder trees. The parameter m shows the number of input bits for each CAM unit, and k indicates the total number of CAM units. If the descriptor has 256 bits and $m = 4$, the number of CAM units is $k = 64$. (b) The comparator tree for selecting the maximum value using Comparator units shown in (c). N is the number of key-points in the reference image. (c) The architecture of each Comparator unit which outputs the maximum of its two inputs (Value 1 and Value 2). 79
Figure 6.4 An example of a various number of CAM units and their effect on the number of hits for matching the two 8-bit binary descriptors with only one bit difference. Partitioning with a higher number of CAM units (for example, $k = 4$ in (a) results in a higher number of hits, while the lower number of CAM units (for example, $k = 1$ in (c)) leads to no hit. 81
Figure 6.5 Timing diagram showing the filling of the CAM. After detecting a key-point, three clock cycles are required to fill the CAM with the location of the descriptor corresponding to the detected key-point. 82
Figure 6.6 Timing diagram showing the reading from CAM for three consecutive key-points. At each clock cycle, the pipeline registers are filled with the key-point data. 83
Figure 6.7 The effect of changing summation threshold on precision and recall of matching. 87
Figure 6.8 The effect of changing Hamming distance threshold on precision and recall of matching. 88

Figure 7.1 Pitch, yaw and roll angles. 93
Figure 7.2 Example image frames of various UAV rotations and movements of the Continuing Studies Building at the University of Victoria. 93
Figure 7.3 Overall block diagram of our proposed approach for combination of a circular shifting binary descriptor and CAM matching. 94
Figure 7.4 Key frame selection and image grouping for evaluation of image matching on consecutive images. 95
Figure 7.5 Comparison of mAP of our proposed method with the ORB algorithm. . . . 95
Figure 7.6 Comparison of frame rate vs. number of key-points. 96
Figure 7.7 Example matching of consecutive frames of the Continuing Studies Building images. The green lines show the correct matches. The red lines are incorrect matches. These images are best viewed in color and zoomed in. 98

List of Acronyms

Abbreviation	Definition
AXI	Advanced eXtensible Interface
ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
BRIEF	Binary Robust Independent Elementary Features
BRISK	Binary Robust Invariant Scalable Keypoints
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
DoG	Difference of Gaussians
FAST	Features from Accelerated Segment Test
FED	Fast Explicit Diffusion
FPGA	Field-Programmable Gate Arrays
fps	frames per second
FREAK	Fast Retina Key-point
GPU	Graphics Processing Unit
HOG	Histogram of Oriented Gradients
HIK	Histogram Intersected Kernel
KNN	K Nearest Neighbours
LBP	Local Binary Pattern
mAP	mean Average Precision
ORB	Oriented fast and Rotated BRIEF
SIFT	Scale-Invariant Feature Transform
SVM	Support Vector Machine
UART	Universal Asynchronous Receiver-Transmitter
vSLAM	visual Simultaneous Localization and Mapping

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all who have supported me during my Ph.D. program.

I am profoundly grateful to my supervisors, Dr. David Capson and Dr. Kin Fun Li who provided this opportunity for me. I am grateful for their guidance, support, and patience. I have learned invaluable skills in terms of research, learning, teaching, and personal growth.

I want to express my deepest gratitude to my parents, Shohreh and Gholamreza, and my sister, Pardis, for their unconditional love, support, and encouragement. Your guidance and support have always been the driving force behind my achievements.

Last but not least, I want to express my deepest gratitude to my loving husband, Sina, who is always my rock throughout every challenge.

Chapter 1

Introduction

Computer vision is a field in artificial intelligence which trains computers to understand the visual world. Over the past few decades, computer vision has enabled humans to take advantage of technology in ways that were not imaginable before. Designing computers that can see, process, and interpret images has resulted in many modern applications including automated surveillance, object recognition, object detection, medical image analysis and self-driven vehicles.

Feature extraction is one of the fundamental operations in computer vision. It is one of the main steps in a wide variety of pattern recognition applications. There have been many feature extraction algorithms proposed in the past decade. HOG (Histogram of Oriented Gradients) [1] is one of the commonly used feature extraction methods which has proven to be useful in many computer vision applications, including human detection [1], car detection [2], and general object recognition [3]. As the speed of image processing is important in the applications in which real-time processing is vital (here real-time is defined as processing the current input image and produce the desired output before the arrival of the next input image.), much work has focused on increasing the speed of the HOG algorithm.

Another fundamental operation in computer vision is image matching which is one of the most important steps for various applications such as object recognition and tracking [4], image retrieval [5], image registration and stitching [6], simultaneous localization and mapping [7], and 3D reconstruction [8]. Image matching corresponds the same content from input images that are taken from the same scene, but may be captured in different conditions such as view point, resolution, and scale [9]. Many image matching algorithms have been proposed in the literature, focusing on improving matching performance and speed. Image matching is a challenging task as processing time is an important aspect in many time critical computer vision applications that require input image on the fly.

In this chapter, we introduce our motivations for the proposed work in this dissertation in Section 1.1. We briefly review the HOG algorithm and its applications in Section 1.2. We review the common applications of image matching algorithm in 1.3, and introduce image matching algorithm in section 1.4. Next, we introduce content-addressable memory in Section 1.5 which we later use for key-point matching in this dissertation. We discuss hardware platforms for image processing algorithms in Section 1.6. In Section 1.7, we present an acknowledgement of author's contributions on shared work.

Finally, in Section 1.8, we present our contributions as well as the structure of this dissertation.

1.1 Motivation

As many applications in computer vision have real-time constraints and are implemented as an embedded system, much research has been focusing on the acceleration of computer vision algorithms. In this dissertation we aim to accelerate image matching algorithms by focusing on increasing the speed of the successive steps of image matching and also propose an implementation to increase the speed of the HOG algorithm.

1.1.1 Image matching algorithm

An important aspect in image matching algorithms is the required time for the processing of the input images. Many image matching applications, such as target tracking and visual simultaneous localization and mapping, require real-time processing of input images. Many real-time applications cannot benefit from the state-of-the-art image matching algorithms due to their computationally expensive processing. This has led much of the published research to focus on proposing techniques and methods to improve the speed of image matching algorithms. In this dissertation, we aim to increase the speed of the image matching algorithms so that they can be utilized in more applications in which high speed processing is important. The main focus of this dissertation is on the acceleration of the individual steps of the image matching algorithms including scale-space generation, patch description, and key-point matching while maintaining accuracy similar to well-known methods in the literature. Although the contributions of this dissertation are not directly related to key-point detection, they can be used as a subsequent step to any key-point detection algorithm.

1.1.2 Histogram of Oriented Gradients (HOG)

HOG is a commonly used feature extraction algorithm in many applications. A high number of required computations has resulted in the time consuming execution of extracting HOG features. Hardware acceleration can boost the execution speed of this algorithm. In many applications, the HOG features are followed by a classifier such as Support Vector Machines [10][11] to make a decision based on the extracted features (for example, object detection). As a part of this dissertation, we propose a hardware–software co-design of the HOG and the subsequent support vector machine classifier.

1.2 An Introduction to the HOG Algorithm and its Application

Dalal and Trigs [1] introduced the HOG algorithm in 2005 and over the years, this algorithm has proven to be useful in many object detection applications. The main idea behind the HOG algorithm is to compute gradients as local descriptors and normalize them locally, and then obtain location invariant features which are robust to illumination changes in the image. HOG features have many

applications such as face recognition [12][13], texture classification [14], vehicle detection [15], and human activity recognition [16]. A complete object detection model can be designed by using HOG features coupled with a classifier such as Adaboost [17] or Support Vector Machines [10][11].

In Chapter 2, Section 2.1 we introduce the HOG algorithm and review its FPGA implementations. In Chapter 3 we present our proposed hardware implementation of the HOG algorithm.

1.3 Image Matching Applications

In this section, we briefly review some important applications of image matching.

Object recognition is the task of identifying an object in different images. There are various approaches used in object recognition applications. The main approaches are global object recognition which detects the objects as a whole and local object recognition which represents an object as a set of interest points (key-points) [4]. Image matching plays an important role in object recognition applications.

Object tracking has a wide range of applications such as surveillance. Object tracking refers to following a moving object in different images or video frames. There are various approaches for object tracking such as region tracking, active-contour-based tracking, model-based tracking, and feature-based tracking. A feature based object tracking pipeline includes elements extraction, clustering and feature matching of images, and benefits from image matching algorithms [18] which are the focus of this dissertation.

Simultaneous localization and mapping (SLAM) algorithms have been proposed for obtaining the 3D structure of an unknown environment. The SLAM algorithm was proposed to achieve the control of autonomous robots. In recent years, SLAM-based applications such as computer vision-based online 3D modeling, augmented reality (AR)-based visualization, and self-driving cars are getting attention in the literature. If the input of the SLAM algorithm is only based on images and videos, the algorithm is referred to as visual SLAM (vSLAM)[7]. An image matching algorithm is used in vSLAM applications to find the correspondences between consecutive frames.

3D reconstruction is another application of image matching. 3D reconstruction algorithms use approaches such as structure from motion (SfM) and multi-view stereo (MVS). These algorithms find a correspondence between images to compute the representation of the 3D shape and convert the input images collectively from 2D to 3D. 3D reconstruction algorithms require an adequate number of images with a small baseline viewpoint differences [8]. The output of these algorithms is a 3D model of an object of interest.

Another application of image matching is for processing aerial images. Aerial images usually have higher dimensions and contain more detail than other images. Some higher level applications that process the matches in aerial images are automated piloting, finding target location for aerial unmanned vehicles, remote sensing, image stitching, and preventing collision with obstacles. As a case study, we apply our proposed methods on aerial images and report the results in Chapter 7.

1.4 An Introduction to Image Matching Algorithms

Image matching is the task of finding correspondences between local points of two or more images of the same scene that might be taken from different view points, weather conditions, illuminations and scale. Many image matching algorithms have been introduced in the literature to be robust to changes in scale, rotation, and illumination.

Image matching may be categorized into dense matching and key-point (descriptor) matching based on the generated output [9]. In dense matching, the algorithms provide a proposal match for every pixel in the images. Although dense matching has important applications such as stereo vision, it is computationally expensive.

Key-point matching scans the images to find pixels with features that are more distinguishable than other pixels, and only proposes matches for these pixels. These pixels are called key-points (or interest points). Key-points typically demonstrating changes in an image include corners, edges, and salient regions. The processing and computations in key-point matching algorithms are limited to small patches of pixels which are neighborhood pixels of the key-points. Key-point matching is thus less computationally expensive than dense matching methods.

Key-point image matching algorithms may be divided into area-based and feature-based methods [9]. Area-based methods use the similarity measurement of the image pixel intensity or information of pixel-domain transformation in the sliding windows to match the images. Sliding windows is defined as moving a window of pixels over an image and comparing the window to the corresponding pixels of the image. Feature-based image matching detects salient points (key-points) in the image and use the local features around these points to match the images. Feature-based methods, which are the focus of this dissertation, consist of scale-space generation, key-point detection, patch description, and key-point matching as shown in Fig. 1.1.

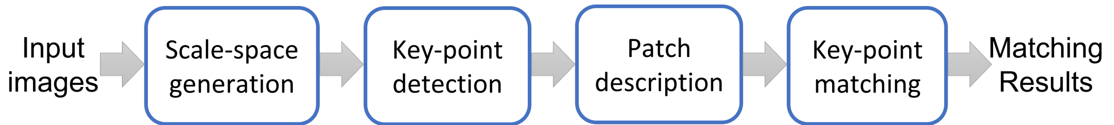

Figure 1.1: Block diagram of feature-based image matching algorithms

In the rest of this section, we introduce these image matching algorithm steps.

Scale-space Generation

Objects in images are found in various sizes and scales. Being robust to scale changes is one of the challenges in image matching algorithms. A common method in matching algorithms to achieve scale invariance is to process the input images in multiple scales including the original images and resized versions of the same images. This idea has led to image pyramids or scale-space representations of an image which are used to replicate the different sizes of an object in images. Input of the scale-space generation step is an input image, and the output is the set of image pyramids of the input image. Each of the images in the image pyramid is the input to the key-point detection step.

In Chapter 2, Section 2.2.1 we review the common scale-space generation algorithms. In Chapter 4, we present our contributions in the acceleration of a non-linear method for the scale-space

generation step of image matching algorithms.

Key-point Detection

Key-point detection is the next step after scale-space generation in image matching algorithms. The goal of this step is to locate a set of key-points (interest points) in the input image. Key-points are usually noticeable parts of an image such as corners or edges that demonstrate variations in pixel values.

In Chapter 2, Section 2.2.2 we review some common key-point detection algorithms.

Patch Description

The next step of the image matching algorithm is patch description. Key-points and their local neighborhoods (local neighbors are also known as ”patches”) are the inputs of this step. Image features are extracted from local patches around the key-points. The feature vector of an image patch extracted in this step is known as the descriptor. For each detected key-point, a descriptor is computed. The computed descriptors are either non-binary descriptors, a vector generated of floating-point values as output, or binary descriptors, whose output is a vector of binary values.

In Chapter 2, Section 2.2.3 we review common description algorithms. We present our contributions on acceleration of the description step of image matching algorithms in Chapter 5.

Key-point Matching

In the patch description step a descriptor is calculated for each key-point. The goal of the key-point matching step is to match each key-point of an image to a key-point of another image. The matching is done based on a distance metric calculated for the descriptors (corresponding to key-points) of the two input images. The output of this step is a set of paired key-points.

We briefly introduce content-addressable memory in Section 1.5. A review of CAM architectures and various implementations are presented in Chapter 2, Section 2.4. In Chapter 6, we present our proposed method for the acceleration of the key-point matching step based on their corresponding binary descriptors, by using a modification to a partitioned content-addressable memory.

1.5 Content-addressable Memory for Key-point Matching

Content-addressable memory (CAM), also known as associative memory, is a type of memory that is used for high-speed search applications. Using CAM as a memory structure in a search application increases the speed as it can identify the location of a query data without any iteration.

In traditional random access memory structures, an address (location) is given as an input to the memory and the corresponding data at that address is read as the output of the memory. Unlike random access memory structures, CAM identifies the location of a query data input. Figure 1.2 depicts an example of the difference in functionality of random access memory structures and CAM.

We review CAM architectures and implementations in Chapter 2, Section 2.4. We introduce our proposed modification to use CAM for key-point matching in Chapter 6.

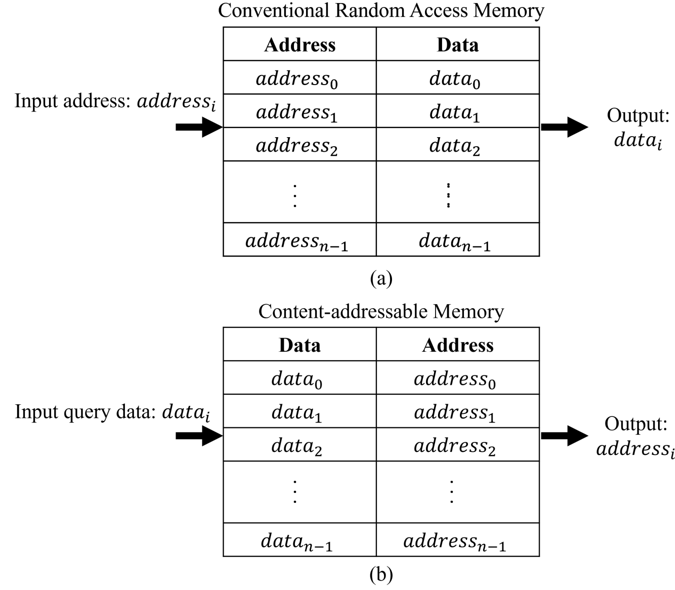

Figure 1.2: Comparison of the functionality of (a) random access memory structure and (b) CAM structure.

1.6 Hardware Acceleration of Image Processing Algorithms

In this work, our focus is on feature-based image matching algorithm and HOG algorithm.

There are many techniques proposed in the literature to increase the speed of image matching algorithms. Some approaches focus on the algorithm level. As many computations of image matching algorithms can be done in parallel, one approach is the implementation of specific hardware circuits for various parts of the image matching algorithm to increase speed.

The HOG algorithm has shown outstanding detection capacity, however it is computationally expensive and requires extensive operations to extract the features of a single frame. Due to this large amount of computation, its software implementation on a stand-alone Central Processing Unit (CPU) may not meet performance expectations. Therefore, there have been many efforts to implement the HOG algorithm (and its variants) on parallel hardware platforms.

The four well-known platforms for implementation of computer vision algorithms are the CPUs, GPUs, FPGAs, and ASICs. Each of these platforms has its own advantages as shown in Fig. 1.3.

Conventional processors or CPUs usually execute instructions sequentially. GPUs introduce some levels of parallelism which can accelerate the implementation of computer vision algorithms. However, there are typically some restrictions in memory access and the number of processes that can be executed on GPUs. Field programmable gate arrays (FPGAs) are popular platforms for implementation of computer vision algorithms due to their capability for parallel computations, reconfigurability, and low power consumption. The designs on FPGAs are optimized and specific to the algorithms and can be integrated alongside a conventional CPU. Application-specific integrated circuits (ASIC) have the lowest price after mass production. However, designing the non-reconfigurable circuits for ASIC requires a high level of expertise and cost.

Figure 1.3 summarizes the trade-offs among the hardware platforms. We choose FPGAs as the platform for our implementations in this work to benefit from custom design and parallel processing.

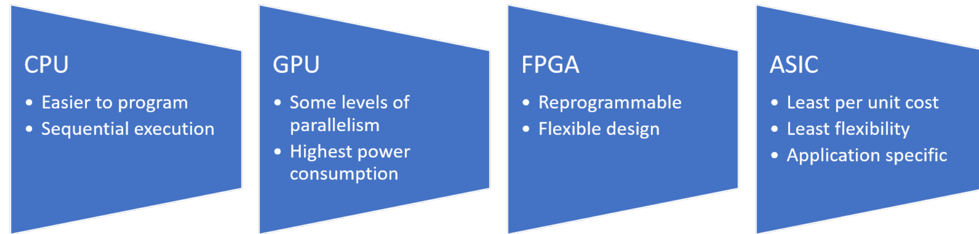

Figure 1.3: The trade-offs in using various platforms for the implementation of computer vision algorithms.

1.7 Acknowledgment of Author's Contributions

This section provides an acknowledgement of the author's contributions to joint authorship under two supervisors. The presented work in [19] analyzes various FPGA-based implementations of the HOG algorithm. In [19], the FPGA-based implementations of the HOG algorithm are categorized in four groups. The first category introduces an analysis on various methods for optimizing the computation of each individual step of the HOG algorithm. The second category presents data manipulation techniques, including numerical representation, data flow modification, and memory optimization. The third category consists of modified HOG-based features with FPGA implementation, and the fourth category discusses hardware-software co-design solutions. The contributions of the author of this dissertation include the third and fourth categories presented in [19].

The survey and analysis of the first and second categories, the guideline design tables, and the overall comparison of all work provided in [19] are the contributions of the other authors.

The work in [20] has four main contributions. First, a new task allocation of the HOG descriptor algorithm to hardware and software platforms is presented. Second, a logarithm-based bin assignment is introduced. The third and fourth contributions are simplifications in the normalization step and parallel histogram generation.

The contributions of the author of this dissertation comprise the coding and implementation of the HOG core and the second contribution (logarithm-based bin assignment). The first, third, and fourth contributions as well as the implementation of the software code and the interface to the hardware circuit are the contributions of the other authors of [20].

1.8 Contributions and Dissertation Structure

This dissertation has resulted in six publications as shown in the following list.

- FPGA-based implementation of HOG algorithm: techniques and challenges, published in *2019 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)* [21].

- Analysis and comparison of FPGA-based histogram of oriented gradients implementations, published in *IEEE Access* in 2020 [19].
- A novel hardware–software co-design and implementation of the HOG algorithm, published in *Sensors* in 2020 [20].
- Real-time FPGA-based implementation of the AKAZE algorithm with nonlinear scale space generation using image partitioning, published in the *Journal of Real-Time Image Processing* in 2021 [22].
- A circular shifting binary descriptor for efficient rotation invariant image matching, published in *2022 26th International Conference on Pattern Recognition (ICPR)* [23].
- A partitioned CAM architecture with FPGA acceleration for binary descriptor matching, submitted for publication.

There are several contributions in this dissertation which are summarized in the following list. These contributions focus on the hardware-software co-design implementation of the HOG algorithm, various steps of the image matching algorithm, including scale-space generation, binary description, and descriptor matching steps. We will introduce each of the following items in more detail in subsequent chapters.

- Analysis and comparison of various methods for FPGA-based implementation of the HOG algorithm [19], [21]: In a literature review, the proposed techniques for the hardware implementation of the HOG algorithm in the last decade are comprehensively compared. Guideline tables are proposed for selecting the methods for each step based on the speed, accuracy and power criteria of the required applications. Due to the high resource usage in these implementations, a hardware-software co-design of the HOG algorithm is proposed which resulted in our next contribution.
- Hardware acceleration of the HOG algorithm, which is a non-binary dense descriptor, using a hardware-software co-design methodology [20]: Our design attains 115 frames per second, which is faster than other published hardware-software implementations, while utilizing fewer resources on chip compared to other work.
- Hardware acceleration of the nonlinear scale-space generation of the AKAZE algorithm [22]: The first step in a scale invariant image matching system is scale space generation. Nonlinear scale space generation algorithms such as AKAZE, reduce noise and distortion at various scales while retaining the borders and key-points of the image. An FPGA-based hardware architecture for AKAZE nonlinear scale space generation is proposed to speed up this algorithm for real-time applications. The three contributions of this work are (1) mapping the two passes of the AKAZE algorithm onto a hardware architecture that realizes the parallel processing of multiple sections, (2) multi-scale line buffers which can be used for different scales, and (3) a time-sharing mechanism in the memory management unit. Approximations in the algorithm are introduced to make hardware implementation more efficient while maintaining the repeatability of the detection. A frame rate of 304 frames per second for a 1280x768 image resolution is achieved which is faster in comparison with other published work.

- A circular shifting binary descriptor is proposed to increase the speed of rotation invariant image matching algorithms [23]: The descriptors of the rotated image patches are computed without rotating either the sample points or the image patch by circularly shifting the binary descriptor. Thus, operations such as multiplication and division from the orientation estimation step are eliminated from the image matching process which significantly reduces the number of operations for computing the descriptor. In addition, our experiments illustrate that the circular shifting binary descriptor shows limited rotation error in comparison with other descriptors such as ORB, while attaining comparable mean average precision.

- A modification on content-addressable memory for binary descriptor matching: CAM is frequently used in high-speed content matching applications. However, due to its lack of functionality to support approximate matching, conventional CAM is not directly useful for image descriptor matching. Our modifications improve the CAM architecture to support approximate content matching for selecting image matches based on local binary descriptors. Matches are based on Hamming distances computed for all possible pairs of binary descriptors extracted from the two images. An FPGA-based implementation of the proposed CAM-based descriptor matching unit is presented to illustrate the high matching speed of our design. The time complexity of our modified CAM method for binary descriptor matching is O(n). Our method performs binary descriptor matching at a rate of one descriptor per clock cycle at a maximum frequency of 102 MHz in our FPGA implementation. The resource utilization and timing metrics of several experiments are reported to demonstrate the efficacy of our design.

In summary, our contributions on hardware acceleration of the HOG algorithm is presented in Chapter 3. We present a hardware design for scale-space generation of the AKAZE algorithm in Chapter 4. For patch description, a novel circular binary shifting descriptor is presented in Chapter 5. We propose a key-point matching method using content-addressable memory in Chapter 6. We also present our experiments on aerial images captured over the University of Victoria by an unmanned aerial vehicle in Chapter 7.

Chapter 2

Literature Review

In this dissertation the focus is on acceleration of the HOG description algorithm and feature-based image matching steps including scale-space generation, description, and key-point matching. In this chapter, first we introduce the HOG algorithm and review its hardware-software implementations in Section 2.1. Then, we review the background of feature-based image matching in Section 2.2. After that, we review hardware implementations for image matching in Section 2.3. Finally, as we used content-addressable memory (CAM) for key-point matching, we review architectures and implementations of CAM in Section 2.4. The content of this chapter is based largely on our publications [19]–[23]

2.1 Survey of Related Work on HOG Algorithm and its Hardware-Software Implementations

In this section, we introduce the HOG algorithm, and Support Vector Machine (SVM) which is a commonly used HOG classifier in Section 2.1.1. We review hardware and hardware-software co-design implementations of the HOG algorithm in Section 2.1.2.

2.1.1 HOG Algorithm and SVM classifier

The HOG Algorithm

The HOG algorithm comprises several steps, as shown in Figure 2.1.

In the first step, the derivatives in the horizontal and vertical directions are calculated for every pixel based on the adjacent pixels around them in a 3 by 3 neighborhood, as shown in (2.1) and (2.2):

$$G_x(x,y) = I(x+1,y) - I(x-1,y) \tag{2.1}$$

$$G_y(x,y) = I(x,y+1) - I(x,y-1) \tag{2.2}$$

where I(x,y) represents the image pixel located in x and y coordinates, and G_x and G_y indicate the

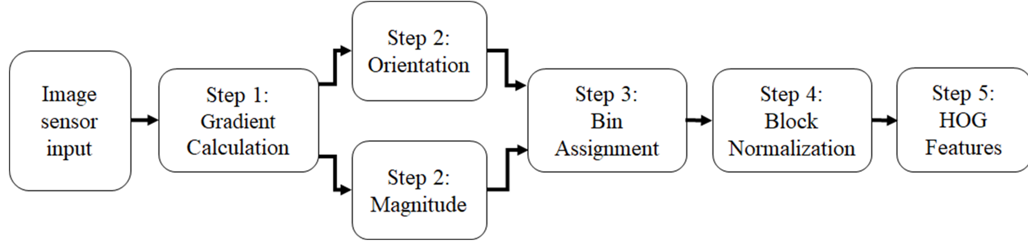

Figure 2.1: A flowchart of the HOG algorithm from input image sensor to HOG features output.

gradients of the horizontal and vertical directions, respectively.

In Step 2, the magnitude of the gradients is computed as shown in (2.3). In addition, the orientation of each pixel is calculated by computing the arctan value of the gradient in vertical direction G_y over the gradient in the horizontal direction G_x, as shown in (2.4).

$$Magnitude(x,y) = \sqrt{{G_x}^2 + {G_y}^2} \tag{2.3}$$

$$Orientation(x,y) = tan^{-1}(\frac{G_y}{G_x}) \tag{2.4}$$

As shown in Figure 2.1, Step 3 adds the magnitude values to the bins according to the orientation of each pixel, for histogram generation. In Step 4, the histograms of the blocks are normalized separately. For block normalization, usually the L2-norm is used. For each block, which contains four histograms, the value of each bin in each histogram is multiplied by itself. The normalized value of each bin is the value of that bin divided by the square root of the summation of the squares of these values, as shown in (2.5):

$$h_n = \frac{h}{\sqrt{\sum |h_i|^2 + \epsilon}} \tag{2.5}$$

where h_n is the normalized histogram, h_i is the value of each bin, h is the initial histogram, and ϵ is a small number to prevent division by zero. The final HOG features are the concatenation of the normalized histograms.

HOG is computed for groups of pixels in the image. For example, every non-overlapping group of 16 pixels (4×4) form a cell and every four cells (2×2) form a block. Figure 2.2 represents this hierarchy. In Figure 2.2, the orientation of the gradient for each pixel is indicated by arrows. The boldness and size of the arrows represent the magnitude of the gradients for that pixel.

Support Vector Machine

We chose an SVM as a classifier for two reasons. First, it is widely used with HOG features and has shown outstanding results, especially for human detection applications [1]. Second, the inference step of this classifier typically consumes fewer hardware resources than other classifiers, such as those based on neural networks. Therefore, after computing the HOG features, we use the SVM classifier for making decisions. An SVM is a linear classifier and is used in many applications. In the

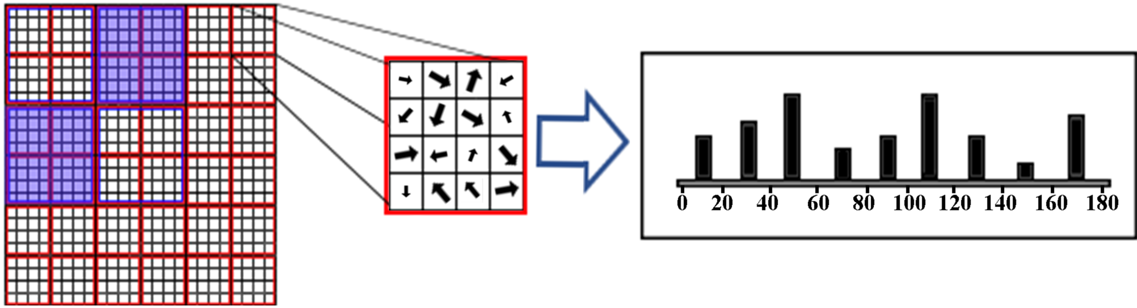

Figure 2.2: Visualization of cells (4 by 4 pixels) and blocks (each containing 4 cells) in the HOG algorithm.

training stage of the SVM classifier, the nearest samples to the decision boundary (support vectors) are determined. Using optimization techniques, this classifier maximizes the margin of the support vectors from the decision boundary. In the testing phase, there is no optimization required. We can classify a sample using only precomputed weights of the SVM from the training stage and the feature vectors, as in Equations (2.6):

$$f(x) = \sum {w_i}^T x + b \tag{2.6}$$

where x represents the input features, and w_i and b are the weights and bias term learned by the classifier in the training stage, respectively. For classifying a sample, $f(x)$ is compared to a threshold (normally zero) and a decision is made based on this comparison. Due to the accuracy and simplicity of the deployment in an SVM classifier, it is a popular choice for hardware implementation.

2.1.2 Hardware–Software Co-design Implementations

We have surveyed different methods for hardware implementation of the HOG algorithm, including an extensive review of methods with hardware–software co-design in our work [19], [21]. In this section, first we briefly review the recent work implementing the HOG algorithm fully on hardware. Then, we review the work using hardware–software co-design methodology.

Hardware Implementation of the HOG Algorithm

There are several implementations of the HOG algorithm using hardware exclusively [24]–[27]. One of the benefits of implementing the HOG algorithm in hardware is of course speed enhancement. Implementing the whole algorithm in hardware is beneficial when resource consumption is not a constraint.

Qasaimeh et al. [25] propose a systolic architecture for hardware implementation of the HOG algorithm. They speed up the histogram generation by reusing the histogram bins generated for adjacent cells. For each sliding window position, they subtract the contribution of the previous column of pixels from the histogram and add the contribution of the next column to the histogram to generate the new histogram values. They speed up their design using this method and achieve 48 fps for 1920×1080 images.

Long et al. [26] propose an ultra-high-speed implementation of the HOG algorithm for object detection. They use a high-speed vision platform which contains a high-speed camera. The vision platform sends 64 pixels per clock cycle to the HOG computation module as input. Instead of storing the HOG values in a memory, they store only the maximum values of the HOG feature vector and its corresponding coordinates so as to simplify computation in subsequent steps.

Ngo et al. [27] propose a long pipeline architecture for the HOG algorithm with 155 stages. Although their proposed system contains a processor, the HOG algorithm is implemented on FPGA. Since they use the processor only for adding bounding boxes onto the output image, we categorize this work as a hardware implementation of the HOG algorithm. In the HOG core, they use the CORDIC (coordinate rotation digital computer) algorithm for computing the magnitude and gradients. At the final stage after computing the SVM score, they convert the fixed-point score value to floating-point and send it to the processor.

Luo et al. [24] propose a pure FPGA implementation of the HOG algorithm. They make several contributions which increase the frame rate of their design. For the bin assignment step, they use a comparison-based method instead of computing the arctangent. This method reduces hardware resource usage, but their design still requires four DSP (digital signal processing) cores for this part. They also propose an architecture for reusing the calculations in the block normalization step and dividing the SVM calculation into partial stages to decrease the overall latency.

Pure hardware implementations of the HOG algorithm have the advantage of higher speed for calculations. Naturally, they consume more hardware resources than systems which assign parts of the tasks to a software processing system. There is a trade-off between the speed of the algorithm and resource utilization, which can be made based on the application and cost evaluation of the processing system.

Hardware–Software Co-design Implementation of the HOG Algorithm

In this dissertation, we focus on the designs which propose a hardware–software co-design approach. The main advantage of these methods is that the resource usage of the hardware can be optimized while preserving the required speed for the application.

Mizuno et al. [10] propose a cell-based scanning scheme for implementing the HOG algorithm. They have parallelized modules for cell histogram generation, histogram normalization and SVM classification. Their proposed parallel architecture increases speed while consuming more hardware resources. Their work is a hardware–software co-design, as they use a CPU to control the HOG algorithm pipeline. They simplify the HOG computation by methods such as using the CORDIC algorithm for gradient calculation, using the Newton method for histogram normalization, and using specific bit-widths for various modules. They store the intermediate data of the histogram of the cells in SRAM memory and load the data for further steps.

Ma et al. [28] propose a hardware–software co-design approach for HOG-SVM computation. They profile code on the CPU to find the most critical and computationally extensive parts of the algorithm. As a result of their analysis, they implement histogram generation and block normalization on an FPGA. They store the result of block normalization in memory, and for the classification step, they re-load the normalized values from the memory. To minimize memory operations, they

store the magnitude and orientation values of each single pixel as a 32-bit value in a single memory location. They propose a multi-scale design which computes HOG for 34 scales. They resize the image and compute the magnitude and gradient in software, and then store the result of this step in the memory on FPGA. In their design, the histogram generation and block normalization steps are assigned to the FPGA, and the results are written back to the memory. After that, the classification module loads the normalized histogram values from the memory and produces the final decision. They also use different bit-widths for different modules, similar to Mizuno et al. [10], so as to have a more efficient implementation.

Rettkowski et al. [29] propose a hardware implementation, a software implementation, and a hardware–software co-design of the HOG algorithm. They implement their design on a Xilinx Zynq® platform, and for their hardware–software model, they use a Linux operating system on the development board. They also compute the histogram generation and block normalization steps on the FPGA. They use SDSoc™ software, which is an IDE (integrated development environment) by Xilinx for implementing heterogeneous embedded systems, to generate hardware modules, and they produce the results for 350×170 pixel windows in their hardware–software implementation. However, for their pure hardware implementation, they process 1980×1020 images and achieve a frame rate of 39.6 fps, which is higher than the frame rate they achieve using their proposed hardware-software co-design of the HOG algorithm.

Huang et al. [30] propose a hardware–software co-design of the algorithm by separating the classification and HOG computation parts. In their design, the HOG generation and computation is done on the FPGA, and the result is sent back to an ARM processor for classification. In order to improve classification, they use the Adaboost classifier first, followed by an SVM classifier to generate the final output.

In the implementation by Ngo et al. [31], the classification step is done on software. They propose a sliding window architecture on hardware for the first part of the HOG algorithm.

Bilal et al. [32] propose a simplification of the HOG algorithm by introducing a histogram of significant gradients. In their proposed method, only the gradients that have a value more than a threshold of average gradient magnitude of a block, cast a binary vote to the histogram. Therefore, there is no need for a normalization step. They use HIK (histogram intersected kernel) (a variation of the SVM) as a classification module, and implement it on a soft processor.

Existing hardware–software approaches have contributed significantly to the state-of-the-art, and research is ongoing to make further improvements. Some of the existing work, such as [10], requires multiple external memory accesses for intermediate results, which can lead to increasing the latency of the design. Another important observation in the existing work is that many include the processor in the flow of the data-path [28]–[31], [33]. This can become the bottleneck of the system, since the processor is usually slower than the programmable logic and processes data sequentially. In [30]–[32], the classification step is assigned to the software side of the system. Since classification is part of the data flow and can start as soon as the first block is processed, assigning it to the hardware part is a better choice to increase the speed of the design.

In Chapter 3, we propose a design which does not require any external memory access for computing an HOG descriptor for each window. We allocate the data-path of the algorithm to the programmable logic and the more simple control loops and address generation task to the processor.

Therefore, the processor does not negatively impact processing speed. In our design, we integrate feature extraction and classification in a unified pipeline to increase the speed of the process.

2.2 Survey of Related Work on Image Matching Algorithms

Image matching is one of the fundamental operations of computer vision for many higher level applications. Feature-based image matching comprises several steps including scale-space generation, key-point detection, patch description, and key-point matching. In this section, we review the common methods for scale-space generation in Section 2.2.1, key-point detection (which is not the focus of this work) in Section 2.2.2, description in Section 2.2.3, and key-point matching in Section 2.2.4.

2.2.1 Scale-space Generation

The first step in a scale-invariant image matching algorithm is scale-space generation. The image pyramid of the input image is generated in this step. Image pyramid or scale-space is introduced to represent the various sizes of an object in images. To create the image pyramid, first the image is resized to generate different scales of the input image. Each of these scales are called an octave. Typically, each octave size is one fourth of the previous octave. After generating octaves, various linear or non-linear filters might be applied to the image of each octave to generate the sub-scales in each octave. The number of octaves and sub-scales depends on the size of the original image and application. Figure 2.3 illustrates an example of scale-space generation.

Scale-space generation methods may be divided into linear and non-linear. In this section we review the scale-space step of the SIFT algorithm as a commonly used linear scale-space generation, and KAZE and AKAZE algorithms as popular non-linear scale-space generation methods.

Lowe [35] introduces linear scale-space generation as the first step of the SIFT algorithm. The SIFT feature detector and descriptor is based on the Difference of Gaussians (DoG) operator. The detector is applied at different scales of an image. In their method, first, the image is resized to generate different octaves. In the next step, Gaussian kernels are applied to the image of each octave.

Alcantarilla et al. [36] propose the KAZE algorithm as a new descriptor with non-linear scale-space generation. They mention that Gaussian blurring does not respect the natural boundaries of objects and smooths to the same degree in both details and noise, thus reducing localization accuracy and distinctiveness. A non-linear scale-space can be generated by non-linear diffusion filtering. In this way, the blurring will be locally adaptive to the image data. The filtering reduces the noise but retains the object boundaries. A superior localization accuracy and distinctiveness is thus obtained. The KAZE algorithm is scale-invariant, and has more distinctiveness at various scales, but it is slower in comparison with linear algorithms. To overcome this drawback, the Accelerated KAZE (AKAZE) [37] algorithm was proposed in 2013. AKAZE non-linear diffusion filtering is based on a Fast Explicit Diffusion (FED) framework which is more efficient in comparison with KAZE filtering.

Although AKAZE is faster in comparison with the KAZE algorithm, it is still slower than linear methods. In Chapter 4 we propose a design to accelerate the non-linear scale-space generation of the AKAZE algorithm.

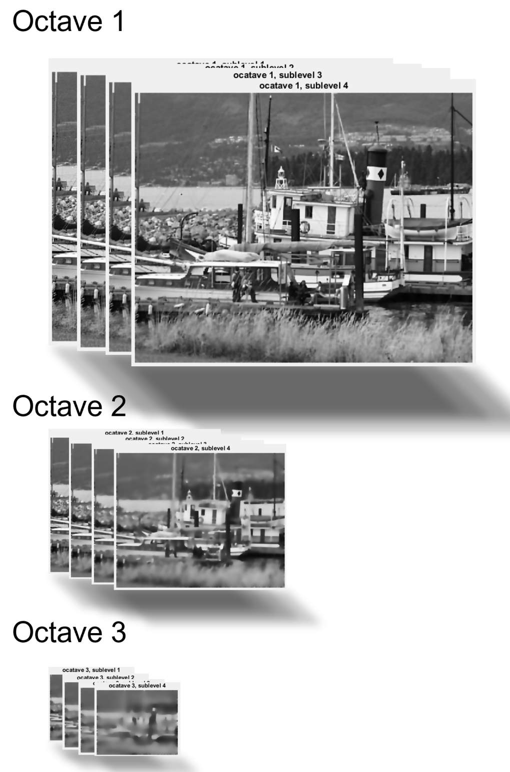

Figure 2.3: An example of linear scale-space generation with 3 octaves and 4 sub-scales in each octave. Images are from the Oxford affine covariant features dataset [34].

2.2.2 Key-point Detection

Key-point detection (detector) is an important step of feature-based image matching algorithms. In key-point detection, repeatable points of interest in the image are detected. In this section, we review the common key-point detection algorithms.

Lowe [35] proposes the difference of Gaussian (DoG) method for key-point detection in the SIFT algorithm. In the key-point detection step, each of the frames in an octave of the scale-space (generated in the scale-space generation step) is subtracted from adjacent frames. The new frames are called the difference of Gaussian frames. The pixels with maximum or minimum values in these frames are selected as key-points.

SURF (Speeded-Up Robust Features) was introduced by Bay et al. [38] to reduce the computational cost of SIFT. They propose Hessian matrix key-point detection as a step for the SURF algorithm. SURF uses the determinant of a Hessian matrix in its key-point detector and takes advantage of integral images to increase the speed of detection. In their method, the Hessian of a pixel in the image is computed using convolution of the Gaussian second order derivatives in horizontal and vertical directions. After computing the Hessian matrix, the determinant of the Hessian matrix is computed. In the next step, this determinant value is compared with a threshold to detect key-points.

Although DoG and Hessian methods have been commonly used in image matching algorithms,

other key-point detection methods have been introduced in the literature to increase the speed. Rosten et al. [39] propose Features from Accelerated Segment Test (FAST) detector. The FAST detector compares each pixel's value with the pixels on a Bresenham circle in the local neighborhood to detect key-points. Figure 2.4 shows an example of pixels used in FAST key-point detection for each patch around a pixel in the input image (potential key-point). The value of the pixel in the center is compared with the 16 pixels on the circle. If 9 successive adjoining pixels have all higher (or all lower) intensity in comparison with the center pixel, the center pixel is proposed as a key-point. In our work we use the FAST detection method for the key-point detection step.

			16	1	2			
		15				3		
	14						4	
	13						5	
	12						6	
		11				7		
			10	9	8			

Figure 2.4: An example of pixels in a patch around a key-point for FAST key-point detection computation. The center pixel is the key-point. The circle around the key-point for comparison are numbered for reference.

2.2.3 Patch Description

After key-point detection, the next step is patch description in which image features are extracted from patches (local neighborhoods) around the key-points. The feature vector of an image patch is also known as the descriptor of that patch. The descriptors of different images are compared in the next steps and based on the comparison, the matching points are selected. The algorithms for extracting features and generating descriptors have great importance in image matching algorithm as they should be robust to rotation and illumination, and describe the patches in a unique way so that similar points in images have similar descriptors.

Ma et al. [9] and Leng et al. [40] have surveyed local descriptors in detail. Local descriptor algorithms may be categorized into handcrafted and learning-based descriptors. Learning-based descriptors learn the parameters of the descriptor by training on a dataset and usually achieve higher accuracy than handcrafted descriptors. Handcrafted descriptors are tailored based on the image characteristics and are typically faster as they require less computation.

Handcrafted descriptors may be categorized into two groups based on the type of output they produce: non-binary descriptors and binary descriptors.

1. Non-binary descriptors

Non-binary descriptors generate a vector of floating-point values as output. Scale-invariant Feature Transform (SIFT) [35], Speeded-Up Robust Features (SURF) [38] are two of the most commonly used non-binary descriptors.

In the SIFT algorithm, after detecting a key-point, a 16x16 patch is extracted and segmented into 16 sub regions. For each sub region, a histogram of gradients is generated. The descriptor is the concatenation of these histograms. The main drawback of SIFT is its computational cost.

In the SURF descriptor, the descriptor is defined by using Haar wavelet responses of the surrounding patch of each detected key-point.

2. Binary descriptors

   Binary descriptors, such as Binary Robust Independent Elementary Features (BRIEF) [41], Oriented FAST and rotated BRIEF (ORB) [42], and LDB [43], output a vector of binary values. The output of binary descriptors is usually generated based on the comparison of local intensities.

   The BRIEF descriptor first pre-smooths the patch around the detected key-point using a Gaussian kernel to reduce the sensitivity to noise. Then, a binary descriptor is generated based on the response of binary comparison tests on sampling pairs and their intensities inside the patch. Rublee et al. [42] propose the ORB algorithm for patch description which is an improvement on the BRIEF algorithm in terms of rotation invariance.

   The Local Difference Binary (LDB) descriptor [43] generates grids by dividing the image patch into 2×2, 3×3, and 4×4 windows and in each window, computes the binary test for the average intensity, the mean of horizontal derivatives and the mean of vertical derivatives. Therefore, for each comparison, 3 bits are generated.

   Alcantarilla et al. [37] propose a modified-local difference binary (M-LDB) descriptor for the AKAZE algorithm. M-LDB uses sub-samples of pixels to compute three bits for each comparison instead of processing all the pixels in the square windows. This makes M-LDB more efficient with respect to the original LDB algorithm [43].

Binary descriptors have several advantages over non-binary descriptors including low memory footprint and faster matching and comparison of descriptors. However, since non-binary descriptors acquire more complex features from the images, they typically result in more accurate matches.

Handling rotation invariance in feature descriptors

One of the complexities of the image matching process is the variation of the reference and target images. Images can have variations such as scale, illumination, and rotation changes. There are different techniques to handle each of these image variations. As we described in Section 2.2.1 scales changes are handled using scale-space generation. Illumination and rotation changes are usually handled in the detection and description steps. Most of the commonly-used descriptors are robust to small illumination changes as they use the gradients or the difference of the intensities in local

patches. However, for higher amounts of illumination changes, Li et al. [44] propose the fusion of camera images with thermal images.

Image descriptors should also be robust regarding rotation changes in images. There are many applications in which the object of interest is rotated in images. Orientation estimation, orientation compensation, and descriptor calculation are typical steps of the patch description step in a local binary descriptor algorithm such as ORB [42] and BRISK [45]. The estimated orientation is used to compensate for the rotation variations which is done typically by rotating either the patch or the sampling patterns. Finally, the descriptors are extracted from the orientation compensated patches.

Figure 2.5 demonstrates an example of patch description steps for an image patch. The steps shown in this figure are applied to all the patches of the reference and target images. In this example, the estimated orientation of the patch from the reference image is $\theta°$. After the orientation compensation step, the patch is rotated so that the estimated orientation of the patch becomes 0. In the next step, the descriptor of the patch is calculated. Finally, the descriptors of all patches are used for the matching step.

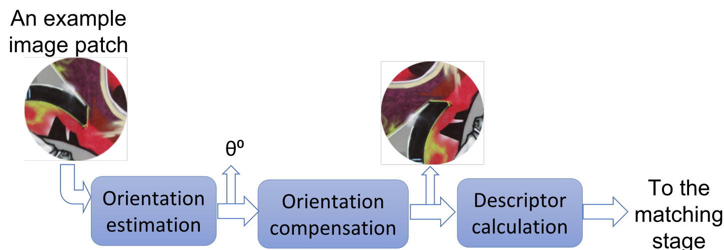

Figure 2.5: An example of orientation handling in conventional patch description algorithms for an image patch. The patch is an image from the Graffiti sequence in the Oxford affine covariant features dataset [34].

In the ORB algorithm, the main orientation of each patch is computed by finding the center of mass of the patch. The main orientation is defined as the vector from the center of the patch to the center of the mass. The location of the center of the mass is computed using the image moments as shown in (2.7):

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \tag{2.7}$$

where the image moments m_{pq} are computed as shown in (2.8):

$$m_{pq} = \sum_{x,y} x^p y^p I(x, y) \tag{2.8}$$

The main orientation is calculated by dividing the moment in the y direction over the moment on the x direction as shown in (2.9):

$$\theta = \arctan(\frac{m_{01}}{m_{10}}) \tag{2.9}$$

After finding the main orientation, the coordinates of the pairs (randomly selected similar to BRIEF) are rotated by the main orientation value so that all patches are in the same direction. Then, the

descriptor is generated similar to the BRIEF algorithm.

Some work such as [46] extract rotation-invariant features from image patches by using polar coordinates.Zhong et al. [47] propose the Rotation-robust Local Difference Binary (RLDB) descriptor. They set a log-polar grid around the patch based on the dominant orientation so that there is no need for orientation compensation before patch description. They generate a binary code by comparing average intensity, radial gradient, and tangent gradient among multiple cells in the log-polar grid around the key-point.

Zhang et al. [48] propose an efficient method for rotation-invariant description of patches. In their method, the image patch around the key-point is divided into co-centeric rings. In each ring, they compute a Local Binary Pattern (LBP) [49] string using a sample point and four surrounding pixels alongside the radius direction from the main key-point. Then, the LBP codes are converted to integer values and the result is used to generate histograms in each co-centeric ring. Finally, the histograms of all rings are concatenated as the final rotation-invariant descriptor. The final features of this method are histograms from co-centeric rings around the key-point and therefore, are not dependent on the position of pixels in the patch with respect to the key-point. As a result, description of a rotated version of a patch using this method leads to a similar descriptor vector as a non-rotated patch.

Bellavia et al. [50] propose the Shifted Gradient Location and Orientation Histogram (SGLOH) as an improvement to the SIFT algorithm for rotation invariance. The advantage of SGLOH is that there is no need to compute the orientation of the patch in the description step. SGLOH divides the image patch into a number of co-centered rings (n_{rings}) where each ring contains a number of sectors ($n_{sectors}$). Therefore, the image patch is divided into $n_{rings} \times n_{sectors}$ distinct regions. Then, the histogram of gradient features is extracted from each region. The bin values of the histograms are obtained by the Gaussian kernel density estimation for each region. Figure 2.6 demonstrates an example of the SGLOH regions for $n_{sectors} = 8$, $n_{rings} = 2$ and 16 regions.

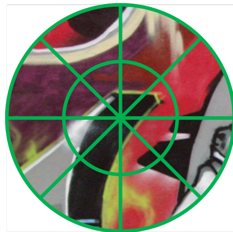

Figure 2.6: An example of regions of SGLOH descriptor. The number of co-centered rings n_{rings} is 2, the number of sectors $n_{sectors}$ is 8, and the number of regions is 16. This figure is based on Fig. 1 in [51].

After generating the histograms in all regions, the SGLOH descriptor is produced by concatenating the histograms as a single vector. The orientation compensation step of the SGLOH algorithm is part of the matching step. For each pair of key-points, the $n_{sectors}$ shifted versions of the SGLOH descriptor of the key-point from the target image are compared to the descriptor of the key-point from the reference image. The shifted descriptor which has the highest similarity to the descriptor of the key-point from the target image is selected to compute the distance metric of the two key-

points. SGLOH simplifies the description process by eliminating the need for computing the main orientation in the descriptor step. Bellavia et al. improve their method in [51] by considering more precise rotations of the patch and thus making the rotation invariance property of SGLOH more accurate.

2.2.4 Key-point Matching

In this step, descriptors are compared across the images (reference image and target image) to identify similar descriptors using a distance metric. The distance metric is calculated among all combination of descriptors for the reference image and target image. Euclidean distance is an example distance metric used for descriptors with floating-point values. Hamming distance is used for binary descriptor matching where each element of the descriptor is a binary value. There are many methods for finding the correspondences from the matching matrix. Some methods use a fixed threshold to select the correct correspondences. Other methods use nearest neighbor algorithms such as KNN [52] to find the correct matches.

In Chapter 6, we introduce a CAM-based solution for key-point matching. Instead of computing the Hamming distance among all combinations of binary descriptors of the reference and target images, our CAM-based binary descriptor matching proposes a nearest approximate descriptor to the query descriptor from the target image. We only compute Hamming distance of the proposed descriptor (as proposed by the modified partitioned CAM) and the query descriptor. If the Hamming distance is less than a pre-defined threshold, the corresponding key-points are considered as a proposed match. In Section 2.4 we review CAM architectures and implementations.

2.3 Survey of Related Work on Hardware Implementations of Image Matching

In this section, we review FPGA-based implementations of the feature-based image matching steps that are related to our work.

2.3.1 Scale-space hardware implementation of the AKAZE algorithm

There are multiple publications that propose accelerators for the AKAZE algorithm. For example, Ramkumar et al. [53] proposed a GPU-based implementation, and Jiang et al. [54] describe a hardware architecture for the AKAZE algorithm based on application specific integrated circuits (ASIC). They achieve a throughput of 127 frames per second for 1920x1080 images. However, their design does not cover the contrast factor calculation which is an essential part of the AKAZE algorithm. The AKAZE algorithm requires two passes through the image and by not implementing the contrast factor, they are eliminating one of the passes which contributes to the higher throughput.

Kalms et al. [55] introduce a hardware accelerator based on FPGAs for extracting AKAZE features. In their initial publication, they propose a pipelined architecture for nonlinear scale space generation and they assume that the contrast factor is computed in software. In their later work

[56], they design an architecture for contrast factor computation as well. They achieve a frame rate of 98 frames per second for a 1024x768 image resolution.

Mentzer et al. [57] propose a hardware accelerator for the AKAZE algorithm based on application specific instruction-set processors (ASIP) which is used for an advanced driving assistance system. They achieve a frame rate of 20 frames per second which is higher than the results obtained from a conventional processor and consumes less power than the FPGAs.

Li et al. [44] use the AKAZE algorithm for extracting descriptors from a video sequence. They use previous frame pixels to predict the first octave of the nonlinear scale space of the current frame in the AKAZE algorithm in order to increase speed. They achieve 784 frames per second for 640x480 images. They propose using motion estimation to reduce the effect of using the previous frame. Based on the published results, this method decreases the accuracy of the algorithm. Their method is beneficial in applications which process high video frame rates in which the amount of changes in successive frames is negligible.

In Chapter 4, we take advantage of the fact that the AKAZE algorithm uses two passes through the input image. For the first pass, we read the image and store it on the FPGA. In the second pass, we process the image in parallel to achieve increased speed. In comparison with [44], our method does not require the previous frames to process the current frame. We achieve a higher frame rate than [56] at the same image resolution and frequency by introducing a memory management unit which facilitates the parallel processing of the image.

2.3.2 FPGA-based implementation of binary descriptor based image matching

In this section we review implementations of image matching which address binary descriptor matching on FPGAs in more detail. There have been many FPGA-based implementations of the detector (key-point detection step) and descriptor (patch description step) steps of image matching algorithms described in the literature [58], [59].

Le et al. [60], [61] propose an FPGA-based CAM-based pattern matching. The objective of [61] is to match a query pattern in an image with a reference image stored in a data base, which has applications such as face detection. However, the focus of [60], [61] is not on the descriptor matching.

Rao et al. [62] propose an FPGA-based implementation of an ORB-based image matching algorithm for full HD videos. After computing the binary descriptors, their binary descriptor matching module calculates the Hamming distance among descriptors stored in block RAMs (descriptors of the reference image) and the descriptors in the data base (descriptors of the target image). Their image matching algorithm requires 13.37 ms for each 1920×1080 frame with 500 features (key-points).

Huang et al. [63] use the BRIEF algorithm for the patch description step of image matching. They also use Hamming distance for the binary descriptor matching step. The maximum number of BRIEF descriptors in their design is 100. The BRIEF descriptors are written to two FIFO memories, one for each image. Then, the Hamming distances of each descriptor from the target image with all descriptors from the reference image are computed in parallel. They use an adder tree for computing the summation in Hamming distance and a comparator tree for finding the minimum distance value from all computed Hamming distances. They achieve 310 fps on the overall system for a 512×512

image frame with 100 descriptors.

Ni et al. [64] propose an FPGA-based binocular image matching algorithm. They use a SURF detector and a BRIEF descriptor with 128 bits. They also use parallel match cores based on Hamming distance for finding the correspondences between left (reference) and right (target) images in a stereo vision system. They achieve 162 fps for 640×480 images.

Peng et al. [65] utilize image matching algorithm for high-resolution aerial images. They use a 512-bit BRISK descriptor and employ 128 Hamming distance calculator modules in parallel. Each module is composed of a 512 bit XOR operation and a bit accumulator to compute the Hamming distance of two binary descriptors. In each clock cycle, they compute the Hamming distance of one binary descriptor with 128 binary descriptors that are pre-stored in SRAM from the description step. Finally, they find the matching key-points associated with two best matches for each key-point using a two-level comparator design. They use high-resolution (5616×3744) images with 4588 pairs which require 548 ms for processing the image matching algorithm.

Hu et al. [66] propose a binary matching system with focus on the symmetry of image patches for achieving high frame rate and ultra-low delay. Their descriptor matching module has three steps. The first step computes the XOR of descriptors in a template and the currently processed descriptor. In the second step, they use population counting for each 32-bit subset of the 128 bits of XOR results in parallel. In the final step (which is called the addition step), the four values of each Hamming distance are added together.

We compare our method with published work which implements image matching algorithms on hardware using conventional binary descriptor matching [62]–[66]. These methods compute a distance metric among all binary descriptors of the two images (reference image and target image). Each query binary descriptor of the target image requires an iteration through each of the binary descriptors of the reference image to calculate the Hamming distance for all possible pairs.

2.4 Survey of Related Work on CAM Architectures and Implementations

Using CAM, we can identify the location of a specific value (query data) in a data memory implemented by a Random Access Memory (RAM), by using that value as an input query data of the CAM. The data stored in CAM is an array of bits where each bit corresponds to a location of the data in the data memory. The most significant bit of this array corresponds to the last memory location in the data memory, and the least significant bit corresponds to the first memory location in the data memory. Figure 2.7 (a) shows a general structure of CAM in which N is the total number of memory locations in the data memory, and N_b is the number of bits. An encoder can be used to encode the hot encoded locations to the binary value of location corresponds to data memory. A numerical example of CAM functionality is shown in Fig. 2.7 (b) in which a query data of 0101 is given to a key register file. The data corresponding to query input is 00000010 is an encoded address, which corresponds to data location 1 in the data memory. If the input query data is 1100, the corresponding value in the key register file is 01100000 which locates two cells in the data memory (addresses 5 and 6). The data in the key register file correspond to 0001, 0011, 0100,

0110, 1000, 1001, 1011, 1101, and 1110 query are equal to 00000000 which means that there is no value equal to those in the data memory (no hit).

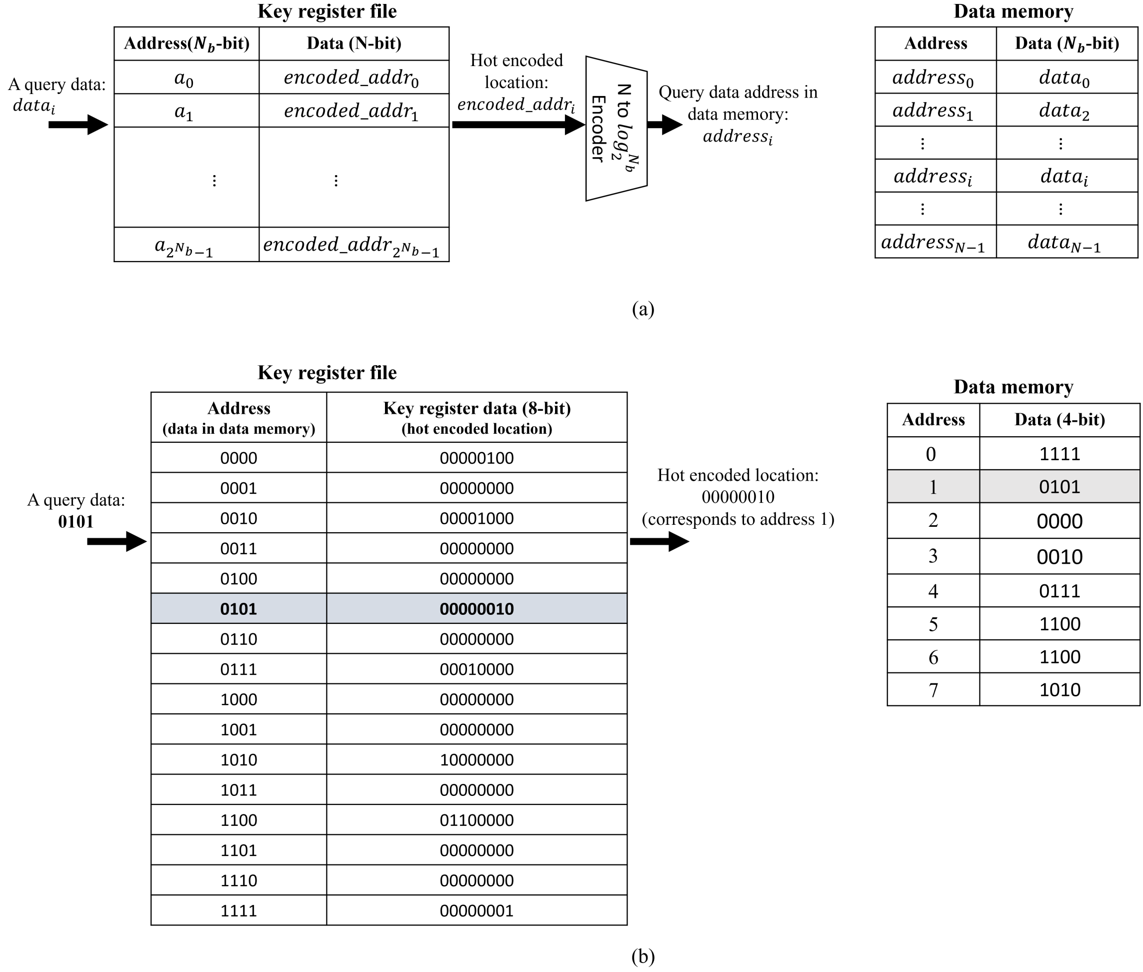

Figure 2.7: (a) A general structure of CAM and (b) an example of finding query data using CAM structure.

Lee et al. [67] propose a nanoelectromechanical-switch-based ternary content-addressable memory (NEMTCAM) for implementing a nearest neighbor classifier. Their proposed NEMTCAM can calculate the Hamming distance by the discharge conductance distribution. Kazemi et al. [68] propose a novel distance function that can be natively evaluated with multi-bit content-addressable memories (MCAMs) based on ferroelectric FETs (FeFETs), to perform a single-step, in-memory nearest neighbor search based on Hamming distance. Garzon et al. [69] propose a Hamming distance tolerant content-addressable memory (HD-CAM) for matching applications. They design the HD-CAM in 65 nm CMOS technology.

Ternary content-addressable memory (TCAM) is an extension of CAM which provides the ability to consider some of the selected input bits as don't-care values in addition to 0s and 1s. TCAM is used when a portion of the input data is sufficient to find the address and an exact match is not

required [70]. In this method the don't-care bit positions in the query data must be specified. TCAM is also not appropriate as the location of don't-care bits should be pre-defined. The disadvantage of the CAM method is its high memory requirement and hardware resources utilization as shown by Irfan et al. [70].

Ullah et al. [71] propose UE-TCAM which partitions the content-addressable memory based on the query data to reduce the memory usage of CAM. The naive implementation of CAM requires 2^{N_b} memory locations, where N_b is the number of bits of data stored in data memory. This configuration, which is shown in Fig. 2.8 (a), results in a large memory footprint for larger N_b.

In UE-TCAM, the query input of CAM is partitioned into a smaller number of bits. Each part of the query input is used to access a separate CAM. As a result, the output of each CAM module shows all the locations in the data memory that have the same bits in the same location of the query input of that CAM module. To find the final location for the query input of CAM, the bit-wise logical AND of all the data read from CAM is calculated and the final non-zero bit provides the location of the query input in the data memory (if all the bits are zero, there is no hit for the query input).

Figure 2.8 (b) is based on the partitioning method in UE-TCAM [71] which presents an implementation of CAM for an input query with N_b bits which are partitioned into k strings of m bits. The data stored in each of the k partitioned CAM units has N bits. In Fig. 2.8, b_{N_b-1} to b_0 represent the query bits, a_{N-1} to a_0 represent the data stored in CAM, and A_{N-1} to A_0 represent the bit-wise AND result of all a_n values in each column. The final AND result is a binary array which is the encoded location of the query data. The position of each 1 value in the binary array corresponds to a memory location in a data memory which contains the query input data.

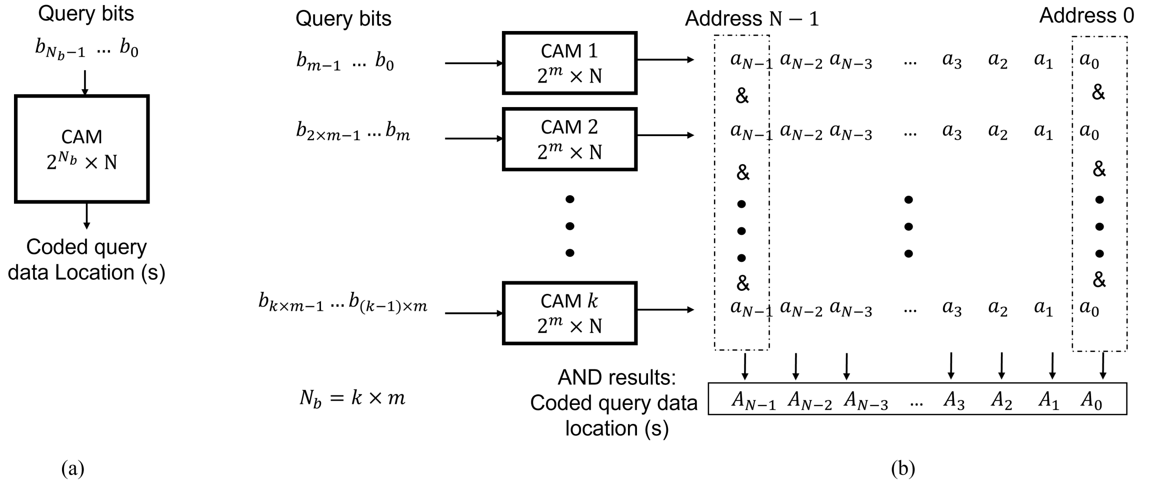

Figure 2.8: (a) Basic implementation of CAM and (b) partitioned implementation of CAM. The total number of bits for the query input (N_b) is partitioned into m-bit address strings for k CAM units. The data stored in each CAM unit has N bits where N is the number of data locations in the data memory.

Irfan et al. [70] have surveyed various implementations of CAM on FPGAs. CAM has been used for many high-speed applications in the literature. Although there are many FPGA-based

implementations of CAM in the recent literature [72]–[75], to the best of our knowledge, there has not been any FPGA-based CAM for solving binary descriptor matching problems. In Chapter 6 we introduce our CAM-based method for binary descriptor matching. We also implement our design on an FPGA to evaluate its speed and resource utilization.

2.5 Conclusion

In this chapter first we introduced the HOG algorithm and briefly reviewed hardware implementation of HOG. Then, we introduced an image matching algorithm and reviewed related work for image matching steps including scale-space generation, key-point detection, patch description, and key-point matching. We also reviewed related hardware implementations of image matching. In the following chapters we present our proposed methods to accelerate HOG implementation and image matching algorithm.

Chapter 3

A Novel Hardware–Software Co-Design and Implementation of the HOG Algorithm

In this chapter, we propose a hardware–software co-design system for the HOG implementation. The content of this chapter is based largely on our publication in *Sensors* [20]. As a case study of the HOG algorithm's application, we choose human detection, which is a real-time application. The INRIA person dataset [76] is one of the more commonly-used datasets for testing human detection approaches. In a real system, the input data would be captured using a digital image sensor, and then converted to grayscale, before being passed to the HOG feature extraction unit. For evaluation purpose, we use the image data from the INRIA dataset for training the SVM classifier and testing our implementation. We validate our design on a Xilinx® FPGA (Kintex® Ultrascale™).

In this chapter we present our hardware-software design to accelerate the HOG algorithm. We introduce our method in Sections 3.1 and 3.2. Our results and comparison with other work are included in Section 3.3. We conclude in Section 3.4.

3.1 Hardware-Software Co-Design of the HOG-SVM System

In this section we introduce our approach to HOG algorithm acceleration. Our contributions are made in two main ways. First are the algorithmic level enhancements, which are the new ideas inside the HOG-SVM core, including logarithm-based bin assignment, block normalization and parallel histogram computation. Second is at the task allocation level, which assigns the appropriate tasks to the processor system and programmable logic of the design.

In a human detection system, a frame of an image is considered as the input. We employ a sliding window technique, as in [1]. We use an 800×600 image resolution and a moving window size of 160×96 on the image. The frame size and the window size are based on the work by Luo et al. [24]. However, it could be readily changed for different applications. We extract the HOG features for all pixels and classify them using an SVM classifier. Since HOG feature extraction and

classification are computationally expensive, we implement the HOG core in hardware in a fully pipelined manner. We allocate the image windowing step to software. This step is responsible for calculating the correct address of the image window in the memory and sending that address to the HOG core.

The main parts of the proposed system are the MicroBlaze™ processor, a DMA (direct memory access) core and an HOG-SVM core. The MicroBlaze™ processor controls the main process by issuing the start signal to the HOG-SVM core and sending the address of an image to the DMA module. We assume that the input image is stored in the BRAM (block RAM) memory, which is the internal memory on the FPGA. This assumption is valid in multiple situations. There are many cases wherein other parts of a computer vision system acquire the image data and have loaded them beforehand in the BRAMs. In addition, since our primary focus is on the architecture of the HOG core, this assumption does not affect the main concept. We read the data from the BRAM in a raster scan streaming mode from the top left of the image to the bottom right. We divide each frame into several smaller windows, which can have overlaps with each other based on the required configuration. For each frame, the processor sends the address of the first pixel of the first row of a window to the DMA. The DMA, which is connected to the memory and the HOG core, reads one row of pixels from memory and sends that row to the HOG core in a streaming channel. The HOG core is designed using fixed-point numbers for efficiency. The core has two AMBA® AXI interface ports. AXI is part of the ARM® advanced microcontroller bus architecture, which provides a parallel high-performance interface. The first interface of the HOG core is based on the AXI light protocol, which is used for communications between the processor and HOG core. The second interface is an AXI stream protocol port which is connected to the DMA for high throughput data transfer. A simplified block diagram of the whole system is shown in Figure 3.1. We use the UART port as a matter of convenience to write the test image in the BRAM memory. Since the BRAM memory can be filled using various methods (depending on the application), this interface could be replaced with another connection interface without affecting the main concepts of this work.

When the DMA is moving data from the memory to the HOG core, one pixel is sent to the core at each clock cycle. There is a finite state machine inside the HOG core to control data receiving and processing. When new data are received, the core processes that data, and in the off times when the processor is sending the address of the next row (or the next window) to the DMA, the core enters a wait state. The whole system described in this section works with a maximum 150 MHz clock frequency. In the next section, we discuss the details of the HOG-SVM core.

3.2 HOG-SVM Core

The overall diagram of the fully pipelined HOG-SVM implementation is shown in Figure 3.2. The solid gray bars represent the registers of the pipeline which we add to reduce the delay of the critical paths. The initial required time for filling the pipeline and generating the first input is $4.25\times W + 14$ clock cycles, where W is the width of the image window. This initial setup time includes $3\times W$ clock cycles in the deserializer module, eight clock cycles in the one-row histogram generator module, W clock cycles in the one-cell histogram buffers module, $W/4$ clock cycles in the two-row histogram buffers module, and six clock cycles for the separation registers shown as gray solid bars in Figure

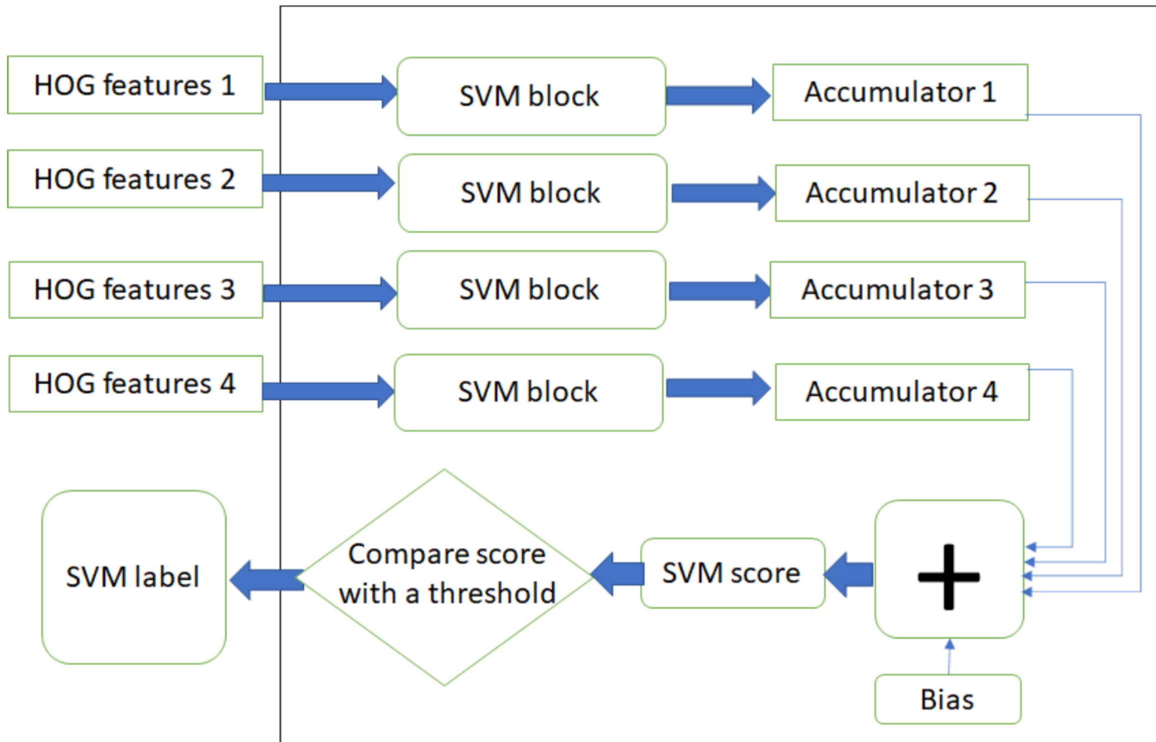

Figure 3.1: Block diagram of the proposed design and port connections

3.2, which are added to reduce the critical timing path of the combinational logic. Gradient and magnitude, and the bin assignment modules, are combinational. The deserializer, one-row histogram generator, one-cell histogram buffers, and two-row histogram buffers all have internal registers and are fully pipelined at the pixel level. When data reach the last stage of the core, all modules work in parallel and there is no need to stop or delay the streaming input in this pipeline. After that, at each clock cycle, one valid SVM output is generated. In this section, we describe the implementation details for each part and the novel contributions.

3.2.1 Deserializer and Buffer Validity Check

The first module of the HOG core is the deserializer unit. This module contains three line buffers which have the depth of the full image window. At every clock cycle, one pixel of the image is read and entered into the first register of the first line buffer, and the values of other registers are sent to the next adjacent registers. For the last register of the first row, the next register is the first register of the second row. Similarly, the value of the last register of the second row is sent to the first register of the third row. After reading three rows of the image, all three buffers are full, and then we can compute the gradients in horizontal and vertical directions.

Figure 3.3 shows the buffers in the deserializer module. The red registers at the end of the line buffer contain the output pixel values of this module, which are sent to the gradient module. The numbers in the first row show the sequence of pixels entering the module. This module requires a setup time of $3 \times W$ clock cycles to fill all registers before producing valid outputs.

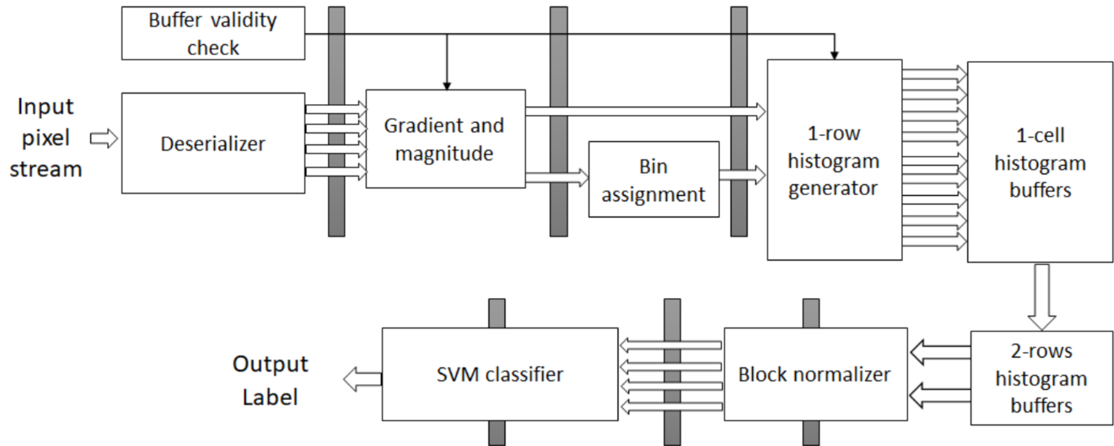

Figure 3.2: The overall diagram of the HOG-SVM core.

Figure 3.3: Line buffers in the deserializer module.

The buffer validity check module is a set of counters which observe the input stream from the deserializer and issue control flow signals, to enable the gradient and magnitude calculation module and the histogram generator module. These signals are important in order to synchronize the flow of valid data in the pipeline.

3.2.2 Gradient and Magnitude Calculation

After deserializing the input stream, gradients in the horizontal and vertical directions are computed in the gradient and magnitude module. The gradient is computed using two subtraction units that subtract the right pixel from the left one and the top pixel from the bottom one. The magnitude of the gradient, which is approximated by the addition of the absolute values of gradients in horizontal and vertical directions, is obtained using two comparators and an adder unit. Since orientation computation and bin assignment are closely related to each other, we design a single unit for this step. Computed gradients are sent to this module for bin assignment.

As mentioned in [19], the original HOG algorithm requires $2 \times W \times H$ multiplication operations (for computing the square of the gradients twice for each pixel), $W \times H$ additions (once for each pixel), and $W \times H$ square root operations (once for each pixel) for computing the magnitude of gradients,

where W is the width and H is the height of the image window. In our implementation, we simplified the magnitude computation by just performing W×H additions (for adding the absolute values once for each pixel) and 2×W×H inversion operations (for absolute value of the gradients twice per pixel).

3.2.3 Logarithm-Based Bin Assignment

In this section, we introduce the new idea of logarithm-based bin assignment. The main advantage of this method is that there is no need to use multipliers, as in [77]. An embedded vision system could have multiple algorithms running simultaneously, and by not using multipliers we can save resources, such as DSP (digital signal processing) cores, for other parts of the system. The idea behind this design originates from the characteristic of logarithm function, which can be used to transform division into subtraction. Equations (3.1)–(3.4) demonstrate the mathematical procedure of this method. Equation (3.1) presents the original orientation computation comparison. In the logarithm-based method, we first compute the tangent of all values as in (3.2). Then, we compute the absolute value and then the base 2 logarithm to all values, as in (3.3). We do not lose any information by computing the absolute value, since we store the sign bit of G_y for choosing the appropriate bin in the next step (we address this in detail later in this section). Subsequently, we separate the dividend and divisor of gradients, as shown in (3.4). We compute the $log_2(|tan(\theta_i)|)$ offline and just calculate the $log_2(|G_x|)$ and $log_2(|G_y|)$ values on the FPGA.

$$\theta_i < tan^{-1}(\frac{G_y}{G_x}) \leq \theta_{i+1} \tag{3.1}$$

$$tan(\theta_i) < \frac{G_y}{G_x} \leq tan(\theta_{i+1}) \tag{3.2}$$

$$log_2(|tan(\theta_i)|) < log_2(|\frac{G_y}{G_x}|) \leq log_2(|tan(\theta_{i+1})|) \tag{3.3}$$

$$log_2(|tan(\theta_i)|) < log_2(|G_y|) - log_2(|G_x|) \leq log_2(|tan(\theta_{i+1})|) \tag{3.4}$$

The reason that log_2 is chosen in this method is because of the bigger slope that this function has in comparison with log_10 or log_e. Figure 3.3 shows the difference in slopes among these functions. The greater the slope of the function is the more differentiable the output is. By using the function with a greater slope, the precomputed logarithm values can then be scaled with a smaller ratio, thus minimizing quantization errors.

To compute the values, we use an LUT-based (Look Up Table) RAM. Depending on the input value which is the computed gradient, we can choose the appropriate log_2 values which are stored in the LUTs of the FPGA.

After retrieving the log_2 values, we subtract $log_2(|G_y|)$ and $log_2(|G_x|)$ from each other, and we can find the appropriate bin based on the subtraction result. To make our design more accurate by taking into account the hardware resource restrictions, we scale the values so that we could prevent mantissa numbers. Equation (3.5) shows the scaled version of (3.4). Both sides of the inequality are precomputed, and for the middle expression, one addition to 160 is added. The reason for adding

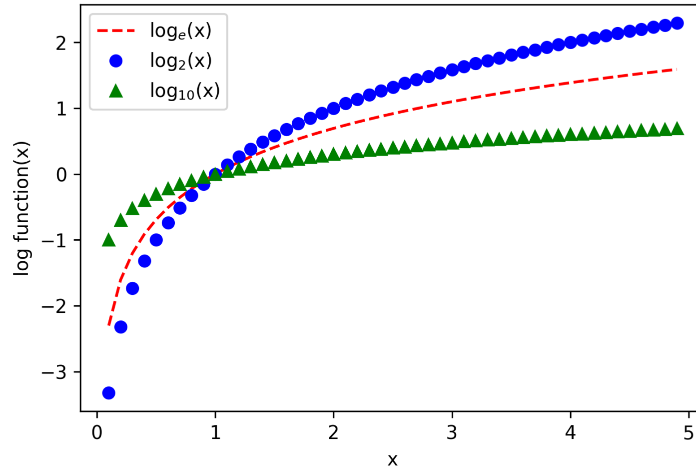

Figure 3.4: Difference between the slope of three logarithm functions.

160 is that we multiply all sides of (3.2) by 32, and then compute the log_2 of them. Then, we multiply the logarithm values by 32. Since $32 \times log_2(32)$ is equal to 160, we add 160 to the middle expression. The LUT based log_2 is to calculate $32 \times log_2$ instead of log2 and only the absolute values of G_x and G_y are given to these LUTs as input.

$$32log_2(32|tan(\theta_i)|) < 160 + 32log_2(|G_y|) - 32log_2(|G_x|) \leq 32log_2(32|tan(\theta_{i+1})|) \tag{3.5}$$

After that, the appropriate bin is selected using the sign bit, as in Figure 3.5. In this figure, L1 to L5 represent precomputed limits for deciding the appropriate bin. After the range of the number is determined, the appropriate bin is selected according to the sign value. In Figure 3.5, v is the term computed by subtraction of the logarithm values. Depending on the sign bit, a range of the orientations is chosen.

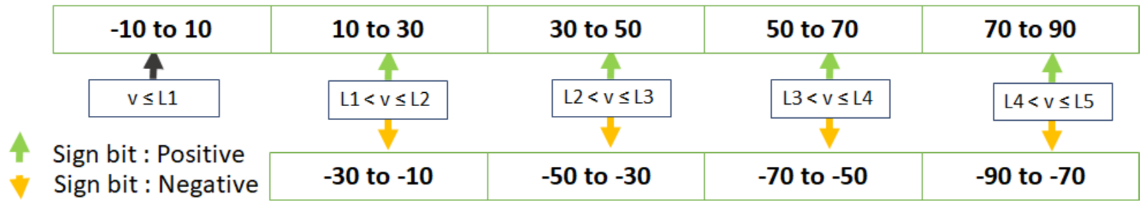

Figure 3.5: The bin assignment procedure.

The limit values in Figure 3.4 are shown in Table 3.1. These limits are precomputed values of $32log_2(tan(\theta_i))$ where θ_i is the bin limit between -90 to 90 degrees.

The pseudo-code for the bin assignment step is shown in Algorithm 1.

It is important to note that for this part, each bin is represented with 12 bits to maintain the accuracy. As mentioned in [19], in the original HOG algorithm, the orientation and bin assignment

Table 3.1: Limits for bin assignment

Limits	Value
L1	80
L2	135
L3	168
L4	207
L5	347

Algorithm 1 The pseudo-code for the bin assignment step

Calculate the absolute values of G_y and G_x
Store the sign of $G_y \times G_x$ in the sign bit
Calculate the scaled logarithms of G_y and G_x
Based on the log value, map to the -90 to 0 degrees bins if the sign bit is negative
Based on the log value, map to the 0 to +90 degrees bins if the sign bit is positive

module require W×H arctangent operations, W×H divisions, and 9×W×H comparison operations. Although some previous works [77] use 18×W×H multiplication operations instead of arctangent, by using our method, the bin assignment module does not use any multipliers. It computes the appropriate bin only by using W×H subtractions (for the $log_2(|G_y|)$ and $log_2(|G_x|)$ subtraction), 9×W×H comparisons (for bin assignment) and 2×W×H inversions (for the absolute value of gradients), and reading values from LUTs. As a result, multipliers and DSP units are saved for other possible processes required in the vision system.

3.2.4 One-Row Histogram Generator

We describe the implementation of a one-row histogram generator unit in this section. This module gets the magnitude and bin assignment inputs from the previous modules. Then, according to the orientation related to each magnitude, a histogram is created for every eight pixels. Computing the histogram requires more than eight clock cycles. This module contains nine registers representing each bin. In the first eight clock cycles, the input enters this module, and the value of each bin is added to the appropriate register, representing an orientation bin. This module requires one clock cycle to output the completed partial histogram, and one clock cycle to reset the registers to zero again to become ready for the next incoming pixels. Since computing histograms in this way requires the input data stream to pause, we design this step by using two partial histogram generators, which work in parallel using a time-sharing protocol. As illustrated in Figure 3.6, the input divider sends a valid magnitude and bin number to the compute histogram modules, and the multiplexer at the end chooses the valid histogram based on the time-sharing protocol. Figure 3.7 demonstrates how the time-sharing protocol works for each eight pixels entering the one-row histogram generator module. Each module requires eight clock cycles to create the histogram, one clock cycle to put it on the output port and one clock cycle to reset the registers. At the 9th clock cycle the output is valid, and at the 10th clock cycle, we reset the registers. While one of the compute histogram modules is in output and reset phase, the other one gets the input stream of data and continues the process. Therefore, there is no need to stop the streaming input. Otherwise, we should pause the streaming input for one cycle for each cell calculation, which could slow a design, especially when processing

high-resolution frames.

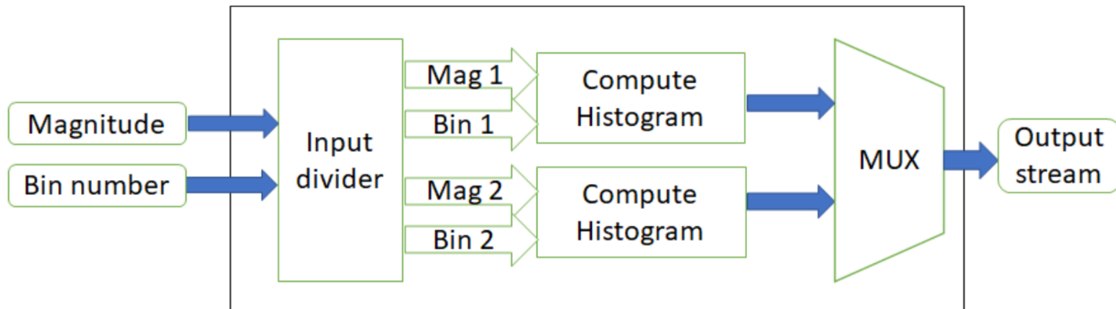

Figure 3.6: One-row histogram generation module.

Figure 3.7: The time-sharing protocol for the histogram generation module.

3.2.5 One-Cell Histogram Buffers

We employ the same architecture proposed by Luo et al. [24] for designing one-cell histogram buffers and two-row histogram buffers. The one-cell histogram buffers module computes the histograms of each eight pixels in a row. The output of this module is nine 16-bit bins of eight pixels in a row every clock cycle. Since our goal is to compute the histogram for 8×8 cells, we use histogram buffers to store the computed histograms sent from the one-row histogram generator module.

This module contains two parts. The first part has eight lines of buffers. Each line has eight buffers. At each clock cycle, a histogram of eight pixels enters this module into the first line buffer. Then, the line buffers work as a shift register, and at each clock cycle, the values are moved through the line buffers. When the first entry of the line buffers reaches the last register, the data in the last register of each line are the histograms of eight pixels of each row of a cell. Therefore, by adding them together bin by bin, we can derive the histogram of a cell. On the next clock cycle, the histogram of the next cell is computed. This process continues until the cell line in the image is changed. While the line buffers are loading up again, their output is not valid.

Figure 3.8 illustrates the eight line buffers of this module. We use a tree-based adding structure to minimize the critical path of the combinational logic for addition. Since we have eight arguments from eight buffers, the tree-based adding structure will have three levels. Therefore, by using a three-level tree-based adding structure, the histogram of a cell can be computed efficiently. This module requires W/8 clock cycles to fill the first row of the buffers, since the input of this module is a histogram computed for eight pixels. Since there are eight rows in this module, a total number

of W clock cycles is required to fill the buffers of this module and generate the first valid output.

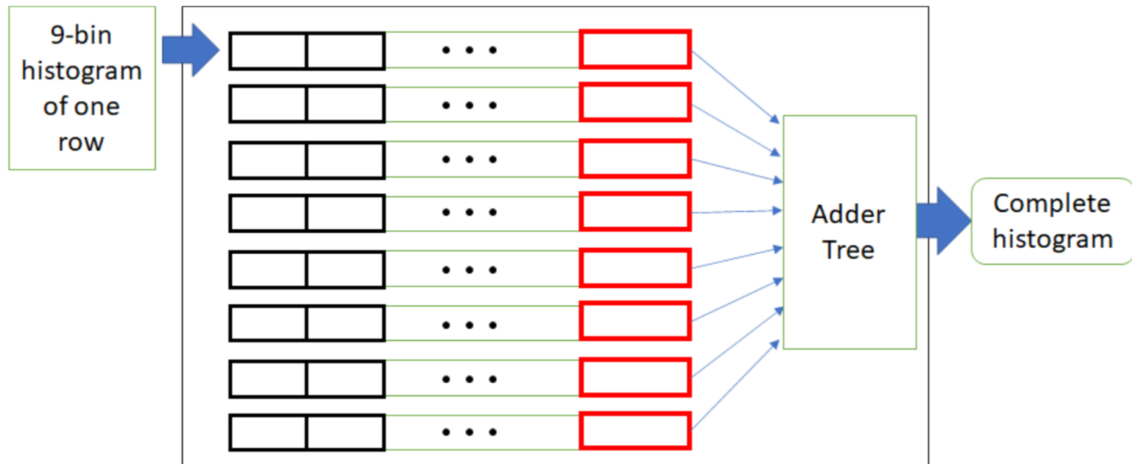

Figure 3.8: The block diagram of one-cell histogram buffers.

3.2.6 Two-Row Histogram Buffers

The next stage is the two-row histogram buffers. The objective of this module is to deserialize the computed cells to have access to four adjacent cells in parallel. Figure 3.9 shows the block diagram of this module. At each clock cycle, if the input is valid, a nine-bin histogram enters these line buffers. When the first cell which has entered this module reaches the last register, we have the histograms of the cells of two cell rows ready at the same time. These values are the output of this module. This module requires a setup time of $2\times W/8$ clock cycles to generate the first valid output, since there are two rows and we have W/8 registers in each row.

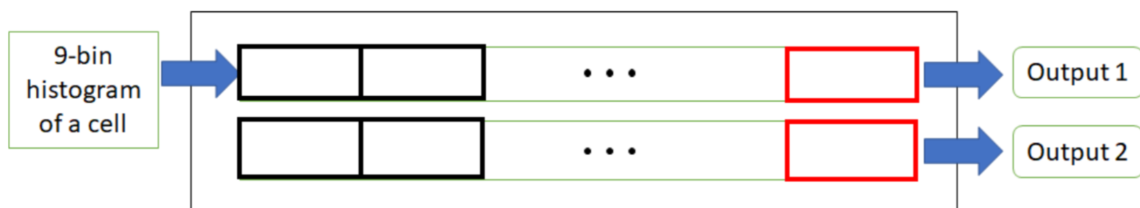

Figure 3.9: The block diagram of two-row histogram buffers.

3.2.7 Block Normalization

The next stage of this design is block normalization. For accurate implementation, if we want to have a latency of one clock cycle for the normalization, 36 multipliers, one square root operation and one division are required. The other possible design is to use one multiplier and compute the square operation once in each clock cycle, which will add 36 clock cycles for the normalization of each block. In this work, we propose a simplified design for the block normalization step. Our design

Table 3.2: Block normalization decoding method

Limits of values for Sum	Bits of the summation	Division of all histogram bins	Number of bits to shift the histograms
Sum > 2047	Bit 11 is checked	Histogram / 16	Histogram >> 4
2048 > Sum> 1023	Bit 10 is checked	Histogram / 8	Histogram >> 3
1024 > Sum > 511	Bit 9 is checked	Histogram / 4	Histogram >> 2
512 > Sum > 255	Bit 8 is checked	Histogram / 2	Histogram >> 1
Sum < 256	—	Histogram	Histogram

normalizes each histogram bin so that the summation of all bins in a block is less than a specific threshold. Choosing a larger value for this threshold will result in less approximation and therefore more accuracy. However, it will consume more hardware resources, since we must dedicate more bits to the result. We choose 255 for this limit as a trade-off between accuracy and resource usage. In addition, we use division by powers of two, which simply shifts the input value and is much less resource-consuming than other division algorithms.

Figure 3.10 shows the block diagram of the normalization module. The block normalization module receives two histograms from two cells in one cell column at each clock cycle, and stores them in the top-left and bottom-left registers. Since the normalization is done for every four cells, this module stores the two inputs for a clock cycle in the top-right and bottom-right registers. In the subsequent clock cycle, when all four histograms are ready, the block normalization module computes the normalized value. First, all bins of the four histograms are added to each other using a tree-based adding structure. Then, depending on the four most significant bits of the sum value, a step-based normalization is adapted using a decoder. Then, each histogram is shifted using a barrel shifter.

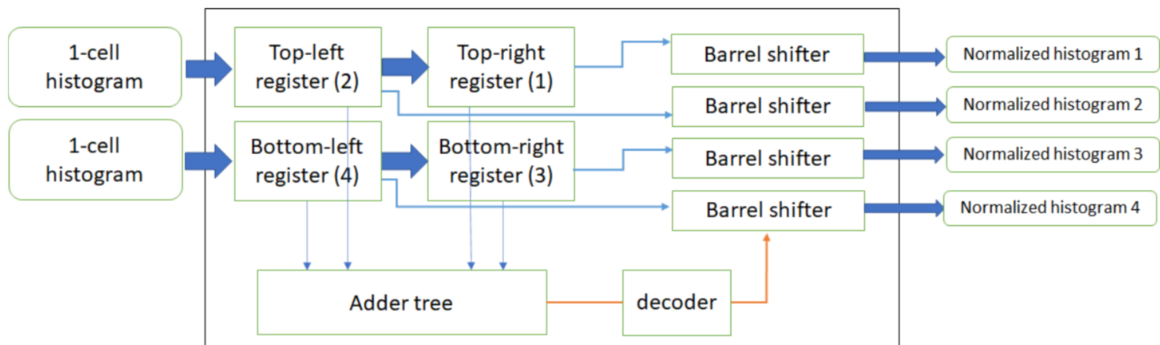

Figure 3.10: The block diagram of block normalization module.

We aim to limit the summation of each block to 255, as shown in Table 3.2. Therefore, depending on the location of the most significant bit that is a '1' in the sum value, all histogram values are divided. If the value of the summation is more than 2047, we divide all histograms by 16. If it is less than that, depending on the bit number, we shift the histograms to the right (each shift divides by two) in order to keep the summation in the range of 0 to 255.

By checking the most significant bits of summation one after another, we can find the range of sum. Based on that, we divide all histograms by shifting the bits to right.

We can benefit from checking one bit of the summation by using binary values for comparison and division. We also perform division by shifting the histogram values, and therefore avoid a complex divider circuit. As mentioned in [19] in the original HOG algorithm, the block normalization step requires 9×C multiplication (for the square of each histogram bin), addition and division operations, and B square root operations, where C is the total number of cells and B is the total number of blocks in an image window. Our simplification results in having 35×B addition operations (for the adder tree) and 36×B shifting operations (for four cells in each block).

3.2.8 SVM Classifier

The last part of the HOG-SVM core is the SVM classifier. In this stage, the output of the block normalization step is given as an input. Since four histograms are normalized at each clock cycle, the SVM module gets four nine-bin histograms as input at once. These histograms are given to the four SVM blocks in this module, as shown in Figure 3.11.

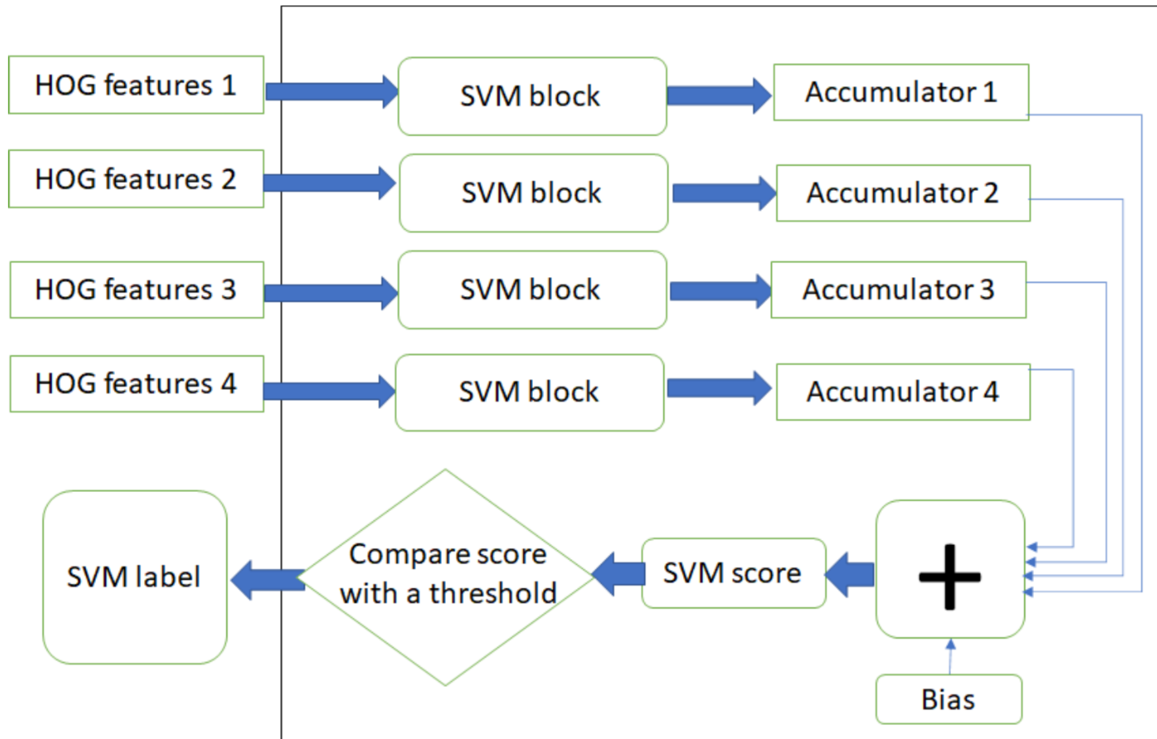

Figure 3.11: The SVM classifier module.

Each of the four parallel SVM blocks contains an SVM RAM, which holds precomputed weights for the SVM classifier. In each SVM block, the input histograms are multiplied bin by bin to the trained weights of the SVM classifier, and their results are added together in four accumulators. The internal logic of an SVM block is illustrated in Figure 3.12. This unit contains an SVM RAM which has the precomputed weights. Nine multipliers are working in parallel in each SVM block module. Finally, when all the data are processed, the values of the accumulators and the bias term of the SVM classifier are added, which is the final score of the SVM classifier. By comparing this score with

a predefined threshold, the SVM will indicate if the image window is a positive or negative sample. If the score is more than the threshold, the label is one, and otherwise, it is zero. In terms of the number of operations, the SVM classifier module requires B comparisons, 36×B multiplications and 40×B addition operations, where B is the total number of blocks in an image window.

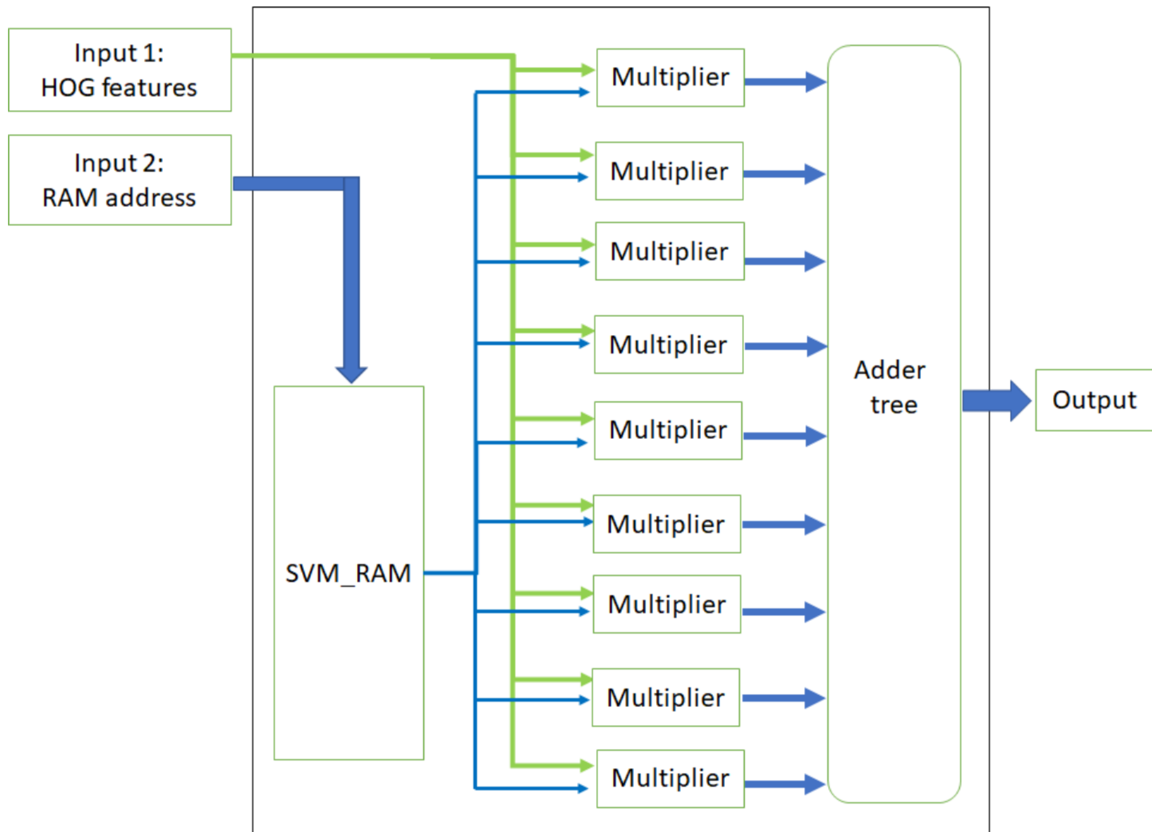

Figure 3.12: SVM block internal logic.

3.3 Results and Comparison with Other Work

Rettkowski et al. [29] were among the first to demonstrate the speed gain of a pure hardware implementation of the HOG algorithm over a software implementation. Pure hardware implementation consumes more resources than when some part of the computation is done on the processor. However, computational approximations in hardware implementations can lead to some accuracy loss. The hardware–software co-design provides a trade-off between preserving the accuracy and limiting hardware resource usage. Therefore, such a design should be compared to other hardware–software designs which are facing the same trade-off in order to have a fair comparison.

Unlike most previous work [28]–[33], in our design, the flow of the data does not include the processor itself. This is important since in those cases, the processor would be the bottleneck of the system. In addition, the HOG-SVM core is designed so that no memory access is required for intermediate computations, as in [10]. Intermediate communications with external off-chip memory

Table 3.3: Comparison with other work

	Reference	FPGA	Image Size	LUTs	BRAM (Kbit)	DSP	Frame Rate (fps)	Pixel per Clock Cycle
Pure Hardware Design	Rettkowski et al. [29]*	Zynq®	1920×1080	41,858	1584	13	39.6	0.99
	Ngo et al. [27]	Cyclone® V	640×480	13,646	317	38	75	0.46
	Long et al. [26]	Stratix® IV	512×512	266,023	47	236	2500	8.19
	Luo et al. [24]	Cyclone® IV	800×600	16,060	334	69	162	0.51
	Qasaimeh et al. [25]	Zynq®	1920×1080	32,871	NA	130	48	0.59
Hardware–Software Co-design	Mizuno et al. [10]	Cyclone® IV	800×600	34,403	334	68	72	0.86
	Ma et al. [28]	Virtex®-6	640×480	184,953	13737	190	68	0.14
	Bilal et al. [32]	Cyclone® IV	640×480	65,501	103	10	25	0.15
	Yu et al. [33]	Spartan®-6	640×480	15,167	351	19	1.5	NA
	Rettkowski et al. [29]*	Zynq®	350×175	NA	NA	NA	0.44	0.0001
	Ngo et al. [31]	Cyclone® V	640×480	12,138	437	65	11	0.02
	Hunag et al. [30]	Spartan®-6	384×288	NA	NA	NA	25	NA
	Our HW-SW co-design	Kintex® UltraScale™	800×600	7804	756	36	115	0.37

*This work did not implement an SVM.

reduce the speed of the system. In our design, everything is buffered using on-chip FPGA resources. Another advantage of our design is that when the first block of the normalized histograms is ready, the classification step starts, and at each clock cycle, one block of data is given to the classifier. Classification is part of the data flow, and assigning it to software as in [32] could decrease performance. At the algorithmic level, we make three contributions. By using the logarithm-based bin assignment, we save four multipliers (DSP units), which can be used for other possible computations or applications on the same chip. By using a simplified block normalizer, we save 36 multipliers, one division unit and one square root operation. In addition, by employing parallel histogram computation, we save 20% of the time for each histogram's generation. We provide the results of our implementation and comparison with other work in Table 3.3. The numbers provided by other work in this table are obtained from their published results.

The last column of Table 3.3 demonstrates the metric of pixel per clock cycle. Our proposed design has a larger pixel per clock cycle value than most of the other hardware–software methods. The work by Long et al. [26] has the highest pixel per clock cycle value. The reason is that in [26], the input of the system is 64 pixels per clock cycle, while others receive one pixel per clock cycle as an input. The work by Mizuno et al. [10], which achieves the highest pixel per clock cycle value in the hardware–software co-design work (due to their highly parallel architecture), uses about twice the number of DSPs and about four times more LUT resources than our proposed design for the same image resolution. In that sense, our design is more efficient in the case of resource usage and, after the initial setup time, can produce a valid output at each clock cycle. Pure hardware implementations are typically faster than hardware–software implementations. However, they will often require more hardware resources as a trade-off.

As shown in Table 3.3, Rettkowski et al. [29] use a Zynq® family FPGA, while Ma et al. [28] use a Virtex® series FPGA. Mizuno et al. [10], Bilal et al. [32] and Ngo et al. [31] use Cyclone® family FPGAs. Cyclone® V devices have more available memory, while Virtex® family FPGAs

have more logic elements than Cyclone® and Zynq® series. The latest FPGAs and technologies should lead to faster systems, however innovative implementation is also a big driving factor in making an effective and efficient system. Ma et al. [28] implement HOG in 34 scales but use the FPGA resources more extensively than other work. The results of Table 3.3 indicate that our system uses a comparable number of DSPs and BRAMs in processing images of similar size, and fewer LUT resources than other work which implement hardware–software co-design systems and pure hardware systems. The frame rate mentioned in Table 3.3 is for the case in which there is no overlap between sliding windows. If we increase the number of overlapped pixels (or decrease pixels stride) the frame rate decreases. Stride is the number of pixels between the current window and the next window in one direction. Figure 3.13 demonstrates the relationship between frame rate and pixel stride in a logarithmic scale. This figure shows that there is a near linear relationship between frame rate and pixel stride.

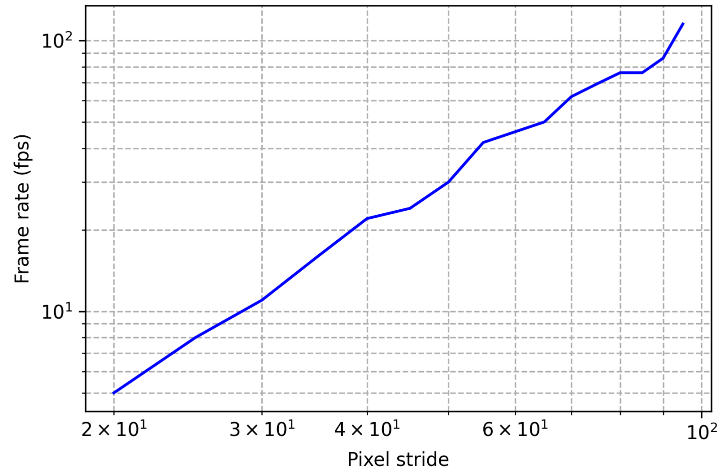

Figure 3.13: The relation between pixel stride and frame rate.

Since the sliding window part of the system only has the responsibility of calculating the correct address of the windows, it is reasonable to choose the processor for this task. On the other hand, HOG and SVM calculations, which require many additions, multiplications and comparisons, are more efficient using hardware. Our design is well-suited for applications, such as mobile and embedded systems, where there is a limitation in hardware resources. By minimizing the usage of hardware resources by HOG and SVM, there are more resources available for other parts of an application, and we can still get accurate and comparable results. Table 3.4 illustrates the resource usage of all parts of the HOG-SVM IP-core.

We present the resource usage of the whole system in Table 3.5. The reset and clock module is responsible for creating the required clock frequencies, and distributing clock and reset signals to all parts of the design. MicroBlaze™ is the main processor, which contains local memory, a debug module, a peripheral controller and an interrupt controller. We use AXI Data FIFO to buffer the streaming information from the DMA module to the HOG-SVM IP-core.

Table 3.4: HOG-SVM IP-core resources

Module name	LUTs	Block RAM Tile	DSP
De-serializer	117	0	0
Buffer validity check	62	0	0
Gradient and Magnitude	8	0	0
1-row histogram generator	502	0	0
One-cell histogram buffers	1960	0	0
Two-rows histogram buffers	376	0	0
Block normalizer	799	0	0
SVM classifier	1622	0	36
Overall	5658	0	36
Percentage used*	1.06%	0	1.87%

*The percentage value is based on the FPGA resources of the KCU105 FPGA board used in this work.

Table 3.5: Resource usage of the whole hardware-software system

Module name	LUT	Block Ram Tile (36Kbit)	DSP
Reset and Clock	17	0	0
Microblaze™	1114	0	0
Microblaze™ local memory	11	16	0
Microblaze™ Debug Module	156	0	0
Microblaze™ Peripheral Controller	179	0	0
Microblaze™ Interrupt Controller	69	0	0
AXI Data FIFO	56	0.5	0
DMA	544	4.5	0
HOG-SVM core	5658	0	36
Sum	7804	21	36
Percentage used*	1.47%	3.5%	1.87%

*The percentage value is based on the FPGA resources of KCU105 FPGA board used in this work.

To measure the speed of the design, we load the input image into the BRAM memory of the FPGA. In our experiments we use 800×600 images so as to be comparable with other hardware–software co-design work, since published results are mostly at this resolution. However, using a higher image resolution such as 1920×1080 does not affect our implementation in terms of resource usage, since the required resources are based on the image window size and not the whole image. The processor starts the computation by instructing the DMA to read from the memory and send the data to the HOG-SVM IP-core. Since in a practical application an external memory can be used and the image can have any arbitrary size, we did not report the number of BRAM memories dedicated to the image stored on the FPGA in Table 3.5, as it is not one of the main elements of the proposed system.

The bandwidth of the designed streaming channel between the memory and the HOG-SVM IP-core is 1.2 Gbit/s, since the DMA can send each pixel in one clock cycle to the HOG core. In our design, for each line of the image, the processor sends a command to DMA to start the data transfer

for a specific number of pixels. Although this controlling mechanism gives the system the capability to process different sizes of the image, it adds an overhead to the timing. Therefore, the data rate of the transfer between the memory and the HOG-SVM IP-core is decreased to 55 Mbit/s, based on our measurements.

In terms of the number of operations, as mentioned in detail in section 3.2, our proposed design has reduced the W×H + 9×C additions, 2×W×H + 9×C multiplications, W×H arctangent operations, W×H + 9×C divisions and W×H + B square root operations in the original HOG algorithm to 2×W×H + 35×B additions, 4×W×H inversions, 9×W×H comparisons and 36×B shifting operations, where C is the total number of cells in an image window. These numbers exclude the parts which were similar, such as the operations required by the SVM module. The SVM module requires B comparisons, 36×B multiplications and 40×B addition operations, where B is the total number of blocks in an image window. We used a hardware model in MATLAB® for evaluating the accuracy of the design. The hardware model produces identical results to the actual implementation on the FPGA. This procedure is similar to the work by Luo et al. [24]. The accuracy results of our system are shown in Figure 3.14. It can be observed that for the test set of the INRIA dataset, the accuracy of our design is very close to, but slightly lower than, that of the software implementation of the algorithm, which is due to the quantization of the floating-point values and simplifications in hardware. Figure 3.14 demonstrates miss rate versus false positive per window, which is the most common method for evaluating human detection systems. The vertical axis shows the miss rate and the horizontal axis represents the number of false positives per window. This diagram is typically drawn in a log–log scale. The software version is our implementation of the HOG-SVM, using MATLAB® software and the Statistics and Machine Learning Toolbox based on [1].

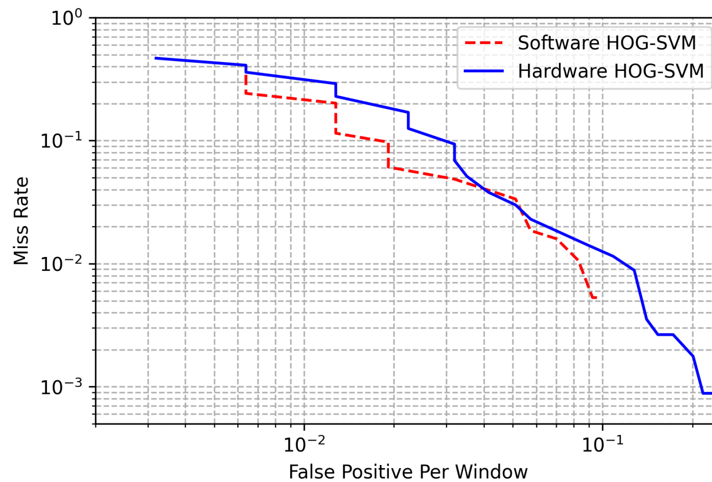

Figure 3.14: Comparison of software implementation and our proposed HW-SW co-design.

3.4 Conclusion

In this chapter, we proposed a hardware–software co-design system of the HOG algorithm, which can receive input data from a digital image sensor, extract the HOG features and make a decision based on those features. Our implementation makes four main contributions. First, at the task allocation level, we propose a well-organized partitioning between different parts in a hardware–software co-design system, which consumes fewer FPGA resources than other comparable hardware–software systems. The idea is to assign the computationally intensive parts of the algorithm, such as gradient and magnitude computation, bin assignment, normalization and classification, to hardware, and delegate the resource-intensive part, which is the windowing stage, to software. Second, as an algorithmic-level contribution, to the best of our knowledge, we are the first to propose a logarithm-based bin assignment in the HOG algorithm, which leads to a multiplier-free implementation of the HOG and reduces the overall number of multipliers for the HOG-SVM core. Third, we propose to use two parallel histogram computation modules, which save one clock cycle for every 8 pixels. As a result, the HOG core can accommodate the pixel data in a streaming manner on each clock cycle without any pause. Finally, we propose a simpler implementation of the block normalization step, which reduces the IP-core resources.

Our design has the capability to use several HOG-SVM IP-cores in parallel for one image. In future, we can modify the design to take advantage of this feature and enhance the speed of the system. Another possibility is to use interrupts efficiently to read precomputed window addresses from the memory. In this way, the processor would have more free time to perform other tasks while the HOG-SVM cores and DMAs are processing the image. Another possible enhancement involves developing other variants of the HOG algorithm and their implementation in hardware. There are many other variants of the HOG algorithm, such as HOG-3d [78], which require a high number of computations and can benefit from parallel implementation.

Chapter 4

Real-time FPGA-based Implementation of the AKAZE Algorithm with Nonlinear Scale-space Generation using Image Partitioning

In this chapter, we propose a hardware design to accelerate the AKAZE algorithm. The content of this chapter is based largely on our publication in the *Journal of Real-Time Image Processing* [22]. In this work, we take advantage of the fact that the AKAZE algorithm uses two passes through the input image. For the first pass, we read the image and store it on the FPGA. In the second pass, we process the image in parallel to achieve increased speed. In Section 4.1, we briefly introduce nonlinear scale-space generation of AKAZE algorithm. The proposed hardware implementation for the AKAZE scale-space generation is introduced in Section 4.2. The analysis of the required timing of the design is described in Section 4.3. The experimental results of our implementation are included in Section 4.4.

4.1 A Brief Introduction to Accelerated KAZE (AKAZE) Non-linear Scale-space Generation

The KAZE [36] algorithm was introduced in 2012 using non-linear scale-space generation. By using non-linear diffusion filtering, the boundaries of the regions in different scales are retained, while reducing noise in the image. Other previous methods find features using a Gaussian scale-space which smooths noise and boundaries of objects to the same degree which results in the loss of detail.

The non-linear scale-space is a set of different scales of the input image. These scales are grouped

as octaves which each of them having four sub-levels in the AKAZE algorithm. Figure 4.1 shows a pseudocode overview of the algorithm for two octaves.

```
Input: Input image   Output: Output images
Preprocessing:
        L_t0 <= Gaussian filter(Input image)
        Compute Kcontrast using input image
for Octave = 1 to 2
        for Sublevel = 1 to 4
                Diffusivity:
                        Compute L_flow based on L_t and Kcontrast for all pixels.
                FED:
                        for j = 1 to N // N varies for different sublevels.
                                Compute L_step using L_flow and L_t
                                L_t <= L_t + L_step
                        end for loop
                Output images { Octave, Sublevel } <= L_t
        end for loop
        Resize image
end for loop
```

Figure 4.1: Pseudocode of AKAZE algorithm

The preprocessing step of the AKAZE algorithm generates a non-linear scale-space. In this step, the image is Gaussian filtered to reduce noise. Then, since the contrast of the image has significant effects on extracting the details of the image, a contrast factor is computed (for use in subsequent steps). In the second step, which computes diffusivity, a conductivity function [37] is calculated using image gradients and a contrast factor found in the preprocessing step. This function affects how much detail of the boundaries of the image is retained in the filtering process. In this chapter, we use the conductivity function [37] in equation 4.1, as follows:

$$L_{flow}(i,j) = \frac{1}{1 + \frac{L_x^2(i,j) + L_y^2(i,j)}{K^2}} \tag{4.1}$$

where K is the contrast factor and L_x and L_y are the gradients of the image computed using a Scharr filter in horizontal and vertical directions, respectively. We use the Scharr filter parameters as shown in Fig. 4.2.

The output of the diffusivity step is called L_{flow} which is computed for each pixel of the image. In the third and final step, which computes the FED, the new sub-level scale is generated using L_{flow} and the previous sub-level. The FED process has multiple iterations (N), the number of which varies depending on the level of the scale-space. The value of (N) for each sub-level is determined using

$$\text{Vertical filter} \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \quad \text{Horizontal filter} \begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix}$$

Figure 4.2: Scharr filter weights

a precomputed array from the original AKAZE algorithm [37]. In each step, a constant step size value is multiplied by the filter.

In each FED process, the summation of the center pixel with four adjacent pixels in vertical and horizontal directions of L_{flow} are multiplied by the difference between the center pixel with four adjacent pixels in vertical and horizontal directions of the previous sub-level. The summation of the results of the multiplications is called L_{step}. The FED calculations are shown in equation 4.2 and 4.3:

$$L_{step}(i,j) = \Sigma(L_{flow}(i,j) + L_{flow}(i+k_1, j+k_2))(L_{t^n}(i,j) - L_{t^n}(i+k_1, j+k_2))s \tag{4.2}$$

with $k_1, k_2 \in \{-1, 1\}$
where L_{step} is the output of the FED calculation, L_t is the previous sub-level and s is the step size constant which is different for each sub-level. The next sub-level is generated as given in equation 4.3:

$$L_{t^{n+1}} = L_{step} + L_{t^n} \tag{4.3}$$

where $L_{t^{n+1}}$ is the value of the next sub-level in the non-linear scale-space.

4.2 Hardware Implementation

Figure 4.3 is the overall block diagram of AKAZE scale-space generation. The main contribution of this work is based on the fact that this algorithm has two passes through the input data. We take advantage of this fact by storing the data in the first pass and process it in parallel in the second pass. We need two memory units for storing the sub-levels (L_t) and the output of the conductivity function (L_{flow}). Each of these two memories has the capacity to store a full image. These two memories are implemented in the Block RAMs (BRAM) of the FPGA. Each BRAM comprises a group of four smaller BRAMs which store a section of an image, divided vertically. The first set of BRAMs contains L_t data and the second set of BRAMs stores L_{flow} data.

This design has three stages. In the first stage, (the preprocessing stage) the 8-bit grey level image enters pixel by pixel to the preprocessing unit in which the contrast factor of the image is calculated, and the image is filtered using a Gaussian blur filter. The contrast factor value is used further in the diffusivity unit, which is the second stage of this design. Then, we store the filtered image, which is the first level of the non-linear scale-space, in L_t memory.

After first stage is completed, the second stage (the diffusivity unit) begins. This unit stores the values in L_{flow} memory in preparation for the third stage, which is FED calculation. From there on, stage 2 and stage 3 work simultaneously until all sub-levels are generated. The output of the third stage is the sub-levels of the non-linear scale-space which are written back to L_t memory for the next iteration. Figure 4.4 shows the data flow of the algorithm at all stages. Further details of each stage are explained in the following sections.

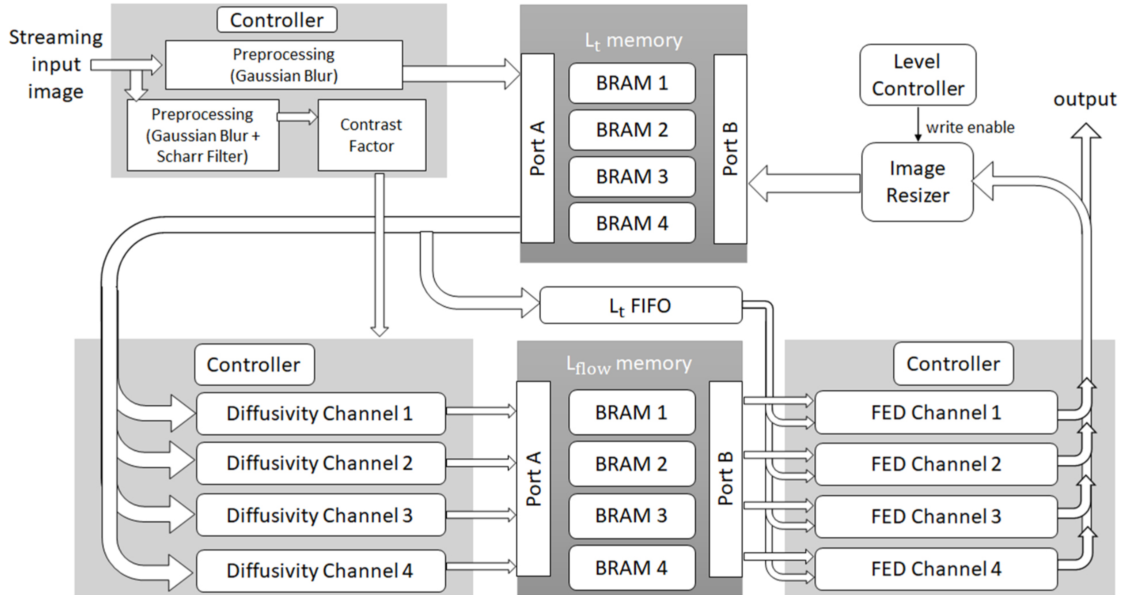

Figure 4.3: Block diagram of AKAZE scale-space generation with four channels

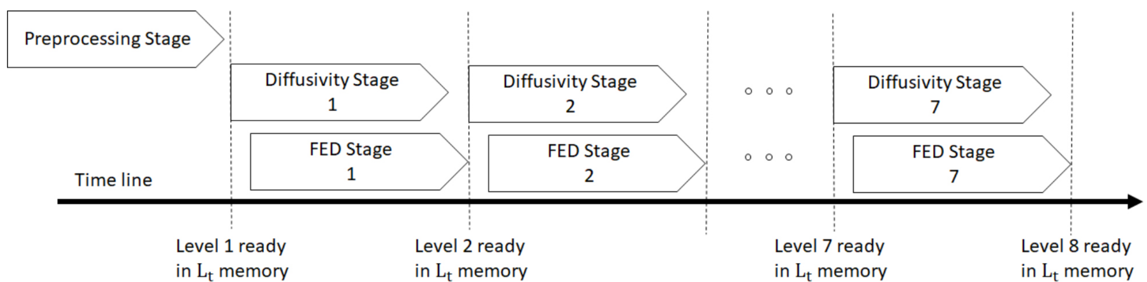

Figure 4.4: Data flow of the algorithm. The FED stage starts after the diffusivity stage. The preprocessing stage only processes the data once at the beginning of the algorithm while the diffusivity and FED stages run in each iteration

4.2.1 Stage 1 - The Preprocessing Unit

The block diagram of the preprocessing unit is shown in Fig. 4.5 This unit has two outputs. The first output is the filtered image, which is the first sub-level and initial value of L_t, and is stored in

the L_t BRAMs. The second output of this unit is the contrast factor of the image, which is used in Stage 2 for the calculation of image diffusivity. To calculate the first sub-level, a 9x9 Gaussian filter is required. The image first enters a line buffer that has a size of Wx9, where W is the image width. The 9x9 window at the end of the line buffer is connected to a Gaussian filter module, in which the filtered value for the center pixel in the 9x9 window is calculated and is stored in the corresponding L_t BRAM memory.

To calculate the contrast factor, first, we apply a 5x5 Gaussian filter to the image. The architecture for this filter is similar to a 9x9 filter and differs only in the size of line buffer and filter module. After filtering the image, the gradients of the image in horizontal and vertical directions are calculated using Scharr filters. The outputs of the Scharr filters are used by the contrast factor calculation module. Finally, the result of the contrast factor module is sent to the diffusivity calculation unit which is the next stage of the algorithm. The block diagram of the contrast factor

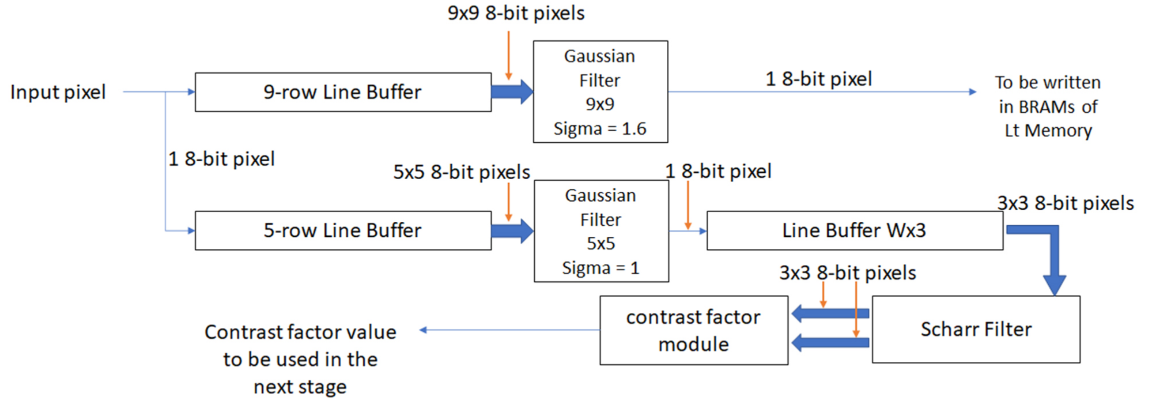

Figure 4.5: Preprocessing stage architecture. This stage contains two Gaussian filter modules. The output of the 9-row line buffer is a 9 by 9 window and the output of the 5-row line buffer is a 5 by 5 window. This stage computes the contrast factor and stores the filtered image in the memory

module is shown in Fig. 4.6. This module receives the horizontal and vertical gradients as input and generates the value of the contrast factor as output. The process of computing the contrast factor value has two phases which is shown in Fig. 6. In the first phase, the value of $L_x^2 + L_y^2$ is computed. In the original algorithm, the square root of $L_x^2 + L_y^2$ is used. However, since this value is used as an address for histogram generation, we can safely set aside the square root. We map this value to 0 to 255 by normalization. This value is used as the address of a set of 256 registers storing the histogram. At each clock cycle, we increment the value of the corresponding register to which $L_x^2 + L_y^2$ is pointing. At the same time, we store the maximum of this value in the Maximum finder register. After this step is finished and the histogram is built, in the second phase, we start from the beginning of the histogram and read the values of the registers and add them in the accumulator. Whenever the value in the accumulator reaches 70% of the maximum value of $L_x^2 + L_y^2$ from phase 1, we store the bin number (same as address value) in the contrast factor register. The value in the contrast factor register is the output of the module.

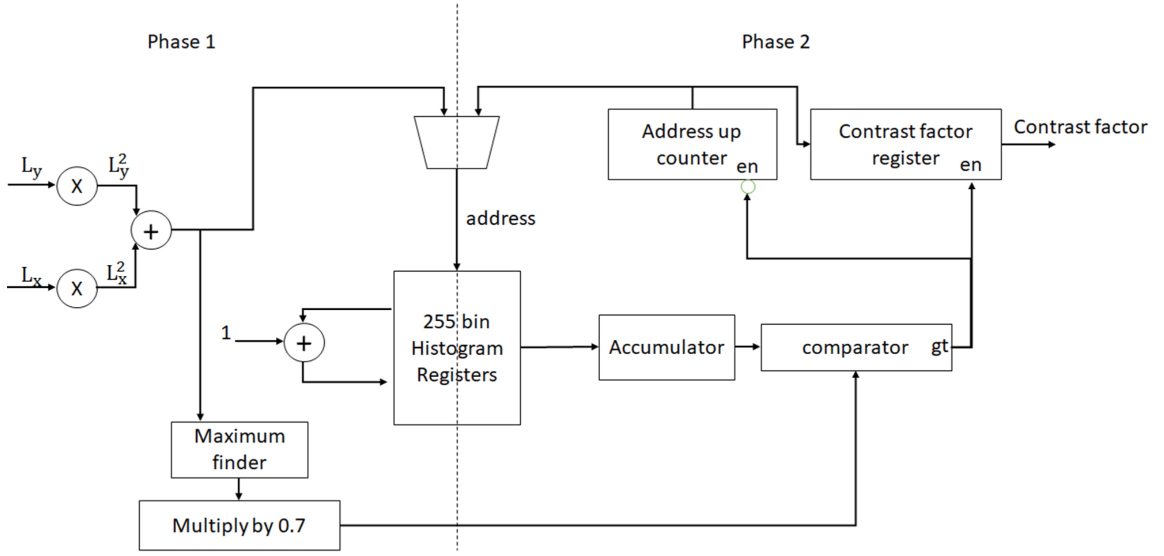

Figure 4.6: Block diagram of contrast factor calculation module

4.2.2 Stage 2 - Diffusivity Calculation

After storing the first sub-level L_{t0} in the L_t memories in the first stage, the second stage, which is the diffusivity stage, begins. Figure 4.7 shows the architecture of a diffusivity channel. In this stage, we read the data from the L_t BRAMs, and the contrast factor value of the image. The contrast factor value is fixed for each image and does not change in the next steps of the algorithm. The L_t data which we read from the BRAM memory enter a 3-row line buffer. The output of the line buffer is connected to two Scharr filters. We compute the gradients of L_t data in x-direction and y-direction using Scharr filters and label them as L_x and L_y, respectively. Then, by using L_x value and L_y value and the contrast factor, we compute the value of L_{flow} according to equation 4.4. For computing L_{flow}, we use a divider IP core provided by Xilinx® [79] which has 43 clock cycles delay. The divisor and the dividend inputs of the IP core are 24-bit and 16-bit integers, respectively. The output of the divider is a fixed-point 40-bit number including 19 fractional bits. We scale the output of the divider to avoid fractional arithmetic. Finally, we store the result of this stage in the L_{flow} BRAMs.

$$L_{flow} = \frac{1}{1 + \frac{L_x^2 + L_y^2}{K^2}} = \frac{K^2}{L_x^2 + L_y^2 + K^2} \tag{4.4}$$

4.2.3 Stage 3 - FED Filtering

In the third stage, we combine the data from L_{flow} and L_t BRAMs to compute the sub-levels in the scale-space. The AKAZE algorithm uses FED filters to generate sub-levels and different octaves. The main processing part of this step is the FED cell module which requires a 3x3 window of L_t data and a 3x3 window of L_{flow} data. In order to prepare the input data for the FED cell in parallel,

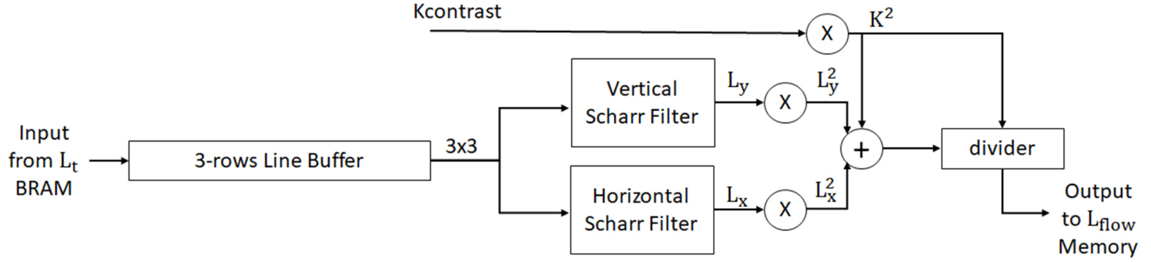

Figure 4.7: Diffusivity channel architecture

we use two 3-row line buffers for L_t data and L_{flow} data, respectively. We compute the output of a FED cell module according to equation 4.2.

The architecture of this module is shown in Fig. 4.8. Each sub-level is generated by the iterative use of FED filters, with the number of FED cells required for each sub-level being different. In this stage, the FED loop is unwrapped to the maximum number of FEDs in the algorithm in order to achieve a pipelined architecture.

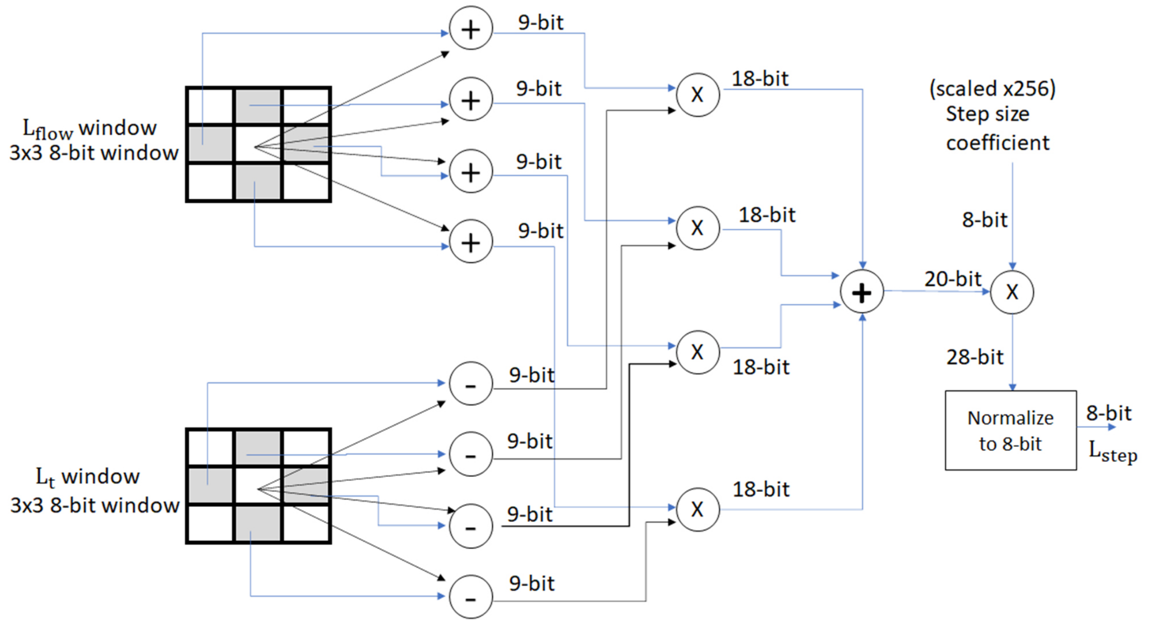

Figure 4.8: FED cell architecture

We label each package of an FED cell and two line buffers as an FED block. Figure 4.9 demonstrates an FED block which generates the output specified in equation 4.3. For generating the first octave, we require four of these FED blocks sequentially, which means that the output of each one is connected to the input of the next. For each sub-level, we extract the output from a specific FED block as shown in Fig. 4.10. A multiplexer is used to select the appropriate output based on the sub-level we are currently generating.

We label each group of 8 FED blocks and the multiplexer attached to them as an FED channel.

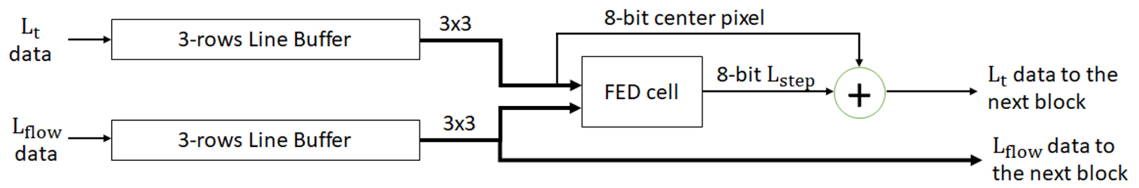

Figure 4.9: FED block architecture which contains two line buffers, an FED cell, and an adder

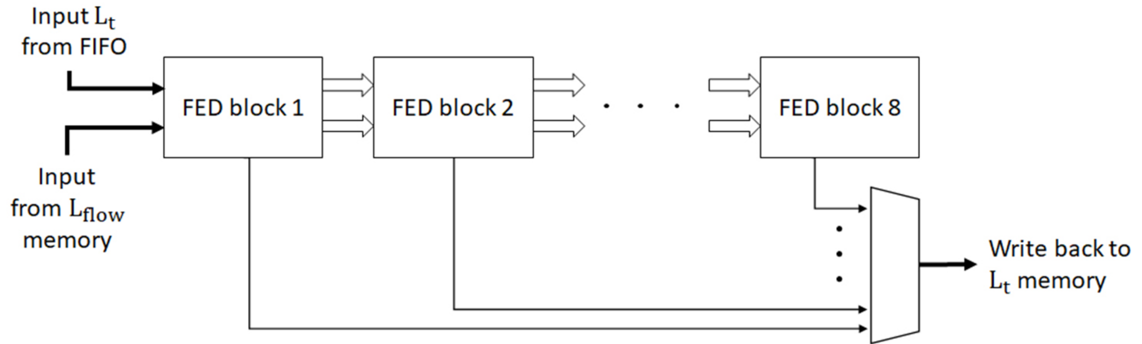

Figure 4.10: FED channel architecture which consists of 8 FED blocks

Since in this design we process the data of the BRAM memories in parallel, 4 FED channels work completely in parallel. Fig. 11. shows the four FED channels. We store the output of the FED channels, which are the sub-level data of the non-linear scale-space, in L_t memory. These data overwrite the previous values of the memory which contains the data from the previous sub-level. At this stage of processing, we have the sub-level data in L_t BRAMs. Now, the diffusivity stage can start again to generate the next L_{flow} for the next sub-level.

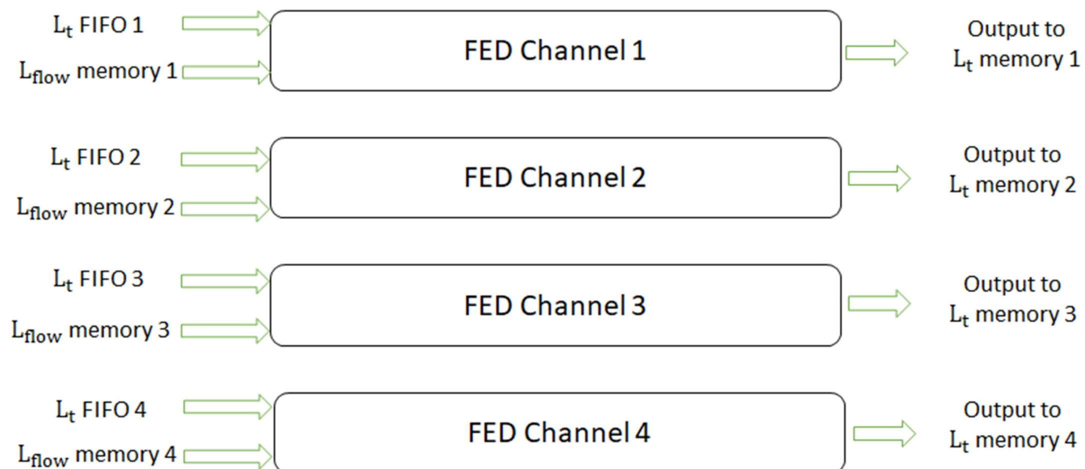

Figure 4.11: 4 FED channels working in parallel

4.2.4 Memory Management Unit

The main contribution in this work is represented in the memory management unit. We have two memories which are dedicated to L_t data and L_{flow} data. The L_t data are the sub-levels of the non-linear scale-space and therefore are the output of the algorithm while L_{flow} data are computed as the required data in the middle of the processing of each sub-level. Each of the memories are divided into n smaller BRAMs (in this design we use n=4), which can be independently written or read. All of these memories are configured as dual port RAMs.

In the first stage (preprocessing) the filtered pixels of the image are written into the four BRAMs of L_t sequentially as shown in Fig. 4.3. The first BRAM is filled and then, the second. This continues until all data are completely read. The algorithm then waits until the contrast factor is computed.

Then, since we have access to all of the image data in the L_t BRAM, we can read from the four BRAMs in parallel. In the second stage of the algorithm, diffusivity channels read the data from the four L_t BRAMs in parallel. Since four diffusivity channels are working in parallel, we can write the data into L_{flow} BRAMs in parallel as well. In our design, we use Port A of the L_{flow} BRAMs to write the L_{flow} values as the outputs of the diffusivity stage. As soon as writing the data is started in the L_{flow} BRAMs, the third stage of the algorithm can start working. In the third stage, FED channels read the data from the L_{flow} BRAMs through port B and process them in parallel. When the output of this stage is ready, it will write back the results into the L_t BRAMs through port B. The architecture of this design is illustrated in Fig. 4.3.

Another key element of the memory management unit is the L_t FIFO between the second and third stages. Since both ports of each L_t BRAM are being used, in order to speed up the design, we use FIFO memories to send the required L_t data from the diffusivity stage to the FED stage. By using a FIFO architecture, we can synchronize the flow of the L_t data and the L_{flow} data to have them available at the same time in the third stage.

Processing the data in each of the n BRAMs separately leads to some undesirable artifacts on the generated output. An example of this artifact is shown in Fig. 12. as black horizontal lines in the image. The reason for this artifact is that the first rows and the last rows of each section require the data of the adjacent rows from previous and subsequent sections, respectively. In order to prevent this artifact, we use a time-sharing mechanism to provide each processing channel with the required data.

In order to prevent the artifacts caused by the border rows in the diffusivity stage, we define three phases for processing each section. There are 4 channels of processing in the diffusivity stage. In the first phase, each channel reads the values from the last two rows of the previous section. As a result, the initial values of the line buffers will be filled with the data from the previous section of the image. In the second phase, each channel reads the data from its own corresponding section in the memory. This phase, which is the main phase of the process, utilizes most of the time of this stage. In the third phase, each channel reads two rows of the data from the next section of the image from the memory. Therefore, the channel has access to the required information from the next section. In order to implement this time-sharing mechanism, we add data multiplexers to the beginning of each diffusivity channel. In addition, we use finite state machines to issue the required control signals for each phase.

Figure 4.12: An example of the artifact from processing four sections of the image in parallel. Image from the Oxford affine covariant features dataset [34]

Since the diffusivity stage and FED stage work simultaneously, when the process in the second phase reaches the last row of a section, the first rows of the next section are already updated with the next sub-level values in the memory. Therefore, we cannot use the current data to prevent the artifact. The solution to this problem is to store the first two rows of each section in another part of the memory and use it in the third phase. We propose a "helping" memory which has the capacity of storing two rows of each section. In each iteration of the algorithm, we fill the helping memories when reading the first two rows of each section in phase two and load from the helping memories of the next section in phase three.

Since the first section of the image does not have a previous stage, the line buffers are filled with zeros in the first phase for the first channel. Similarly, we use zeros as the input data for the last channel in phase 3 since there is no section after that. Therefore, memories 1 and 2 are connected to the diffusivity channel 1 using a multiplexer. Memories 1, 2, and 3 are connected to the diffusivity channel 2 using the second multiplexer. Memories 2, 3, 4 are connected to the diffusivity channel 3 using the third multiplexer and memories 3 and 4 are connected to the diffusivity channel 4 using the fourth multiplexer. We use the same procedure for FED channels and L_{flow} memory to prevent the artifacts. Fig. 4.13. demonstrates the time-sharing mechanism for preventing the line artifacts in the non-linear scale-space.

4.2.5 Image Resizer

In the original AKAZE algorithm, after each octave is generated, the size of the image is reduced by half. In our design, the image resizer module issues the required signals to store only half of the

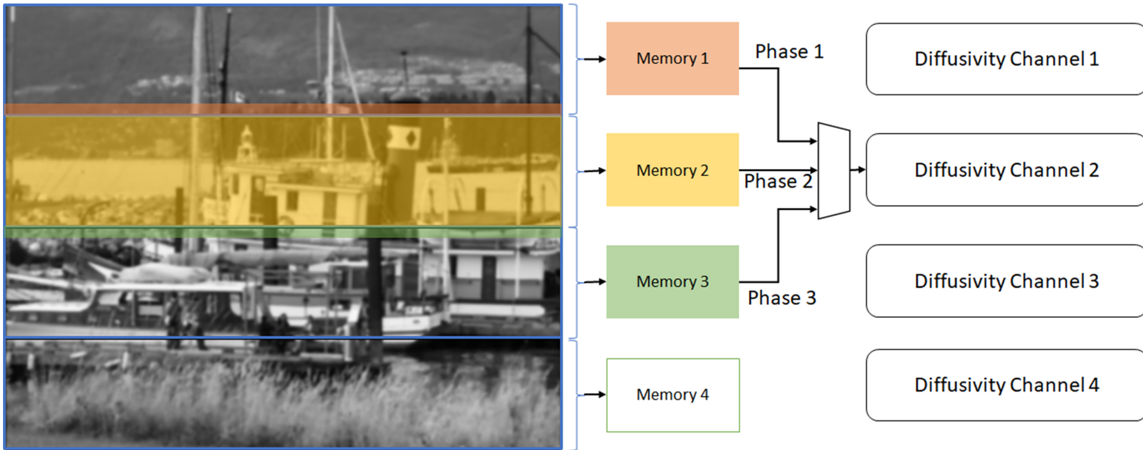

Figure 4.13: An example of selecting three phases for reading data from various sections of the memories. We show the data flow for diffusivity channel 2 as an example. In phase 1, this channel reads the data from the last two rows of the first section of the image. In phase 2, data enter channel 2 from the second section and in phase 3, diffusivity channel 2 reads the first two rows of data from the next section. Other channels have a similar data flow. Image from the Oxford affine covariant features dataset [34]

image in the memory to resize the image. In order to do so, this module controls the write enable signals of the B port of L_t BRAMs. When we are generating the first level of the second octave, the resizer module disables the write enable signal when the FED channels are generating the outputs of even rows and even columns. Therefore, only odd rows and columns are written into L_t BRAM memories and the size of the image is thus reduced by half.

After this step, all other parts of the design work with the smaller image. To do so, we design each of the line buffers in the diffusivity and FED stages to have the capability to work with two sizes. The architecture of the line buffers with three rows is shown in Fig. 4.14. If the line buffer has more than three rows (for example, 5 or 9 rows) the concept is the same and only the number of the registers is different.

The line buffers have two modes. In the first mode, we use the full capacity of the line buffers. The input pixels at the end of each line are written to the beginning registers of the next line. In this mode, the output window is derived from the last registers of each line. This mode is used when we are processing the first scale of the image. The second mode, which is for half scale of the image, the output of the registers in the middle of the original line buffer is sent back to the next line. Therefore, we need to use multiplexers to select the correct input for the first registers of each row. In addition, the output window is derived by the registers in the middle of the line buffer. Therefore, there is also a multiplexer to choose the appropriate window as the output of the module. All of the multiplexers in the line buffers are controlled using a size mode signal which is generated by the level controller module that contains a counter that keeps counts of the sub-levels being generated.

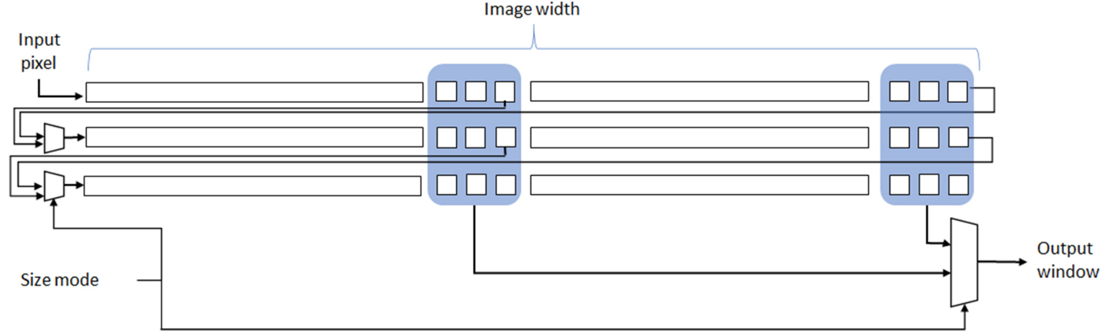

Figure 4.14: The architecture of the 3-row line buffer with multi-scale capability

4.3 Timing Analysis

In this section, we analyze the required timing of the architecture and calculate the throughput of the design, after each line buffer is initialized. This initialization time is needed until the output of the line buffers becomes valid and we can have access to the data of multiple rows in parallel. We use zero padding to process border pixels to avoid reducing the part of the image that we are processing.

In the preprocessing stage, we have a 9x9 Gaussian filter module. Therefore, we need a 9xW line buffer where W is the width of the image. The initialization time required for this stage is 5xW since after 5 rows of the image are read, we can have valid output from this module (other rows are initially 0s). After 5xW clock cycles, the output of the Gaussian filter is valid and after that we need WxH clock cycles to process the whole image. In this estimation we did not include the contrast factor calculation since it overlaps with filtering the data and its overhead is negligible. Therefore, the required time for preprocessing is

$$T_{preprocessing} = 5W + WH = W(5 + H) \tag{4.5}$$

The next stage of the design is the diffusivity stage. In this stage, we first have line buffers for generating a 3x3 windows as inputs for the Scharr filters. These line buffers require 2xW clock cycles for initialization and is the first phase of the time-sharing mechanism. After that, since we are processing the image in n different sections in parallel, we require (WxH)/n clock cycles to read and process n sections of the image. In addition, an initial 43 clock cycles are required for the divider module. After that, at each clock cycle, the divider generates new valid results. Hence, the number of required clock cycles for the diffusivity step is based on the image width, height, and the number of parallel sections according to:

$$T_{Diffusivity} = 2W + \frac{WH}{n} + 43 = W(2 + \frac{H}{n}) + 43 \tag{4.6}$$

The next stage is the FED module. In this stage, similar to the diffusivity stage, we use 3-row line buffers in each FED block module. Therefore, we need 2xW for initialization of each FED block module. In addition, WxH/n clock cycles are required for reading and processing the pixels of each

section of the image. Since for each sub-level we get the output from a different FED block, we do not need to wait for the data to pass all the FED blocks in an FED channel in this stage. The first octave has four sub-levels. The first sub-level is the filtered image and therefore there is no need to compute the result of the FED stage for it. For the second and third sub-levels, we get the outputs from the second FED block and for the fourth sub-level, we get the output from the third FED block. In the second octave, for the four sub-levels of five, six, seven, and eight, we get the output from the third, fourth, fifth and sixth FED block, respectively. It is important to note that for the second octave, the size of the image is reduced to half size and therefore we use W/2 and H/2 as width and height of the image. Hence, the number of required clock cycles for this stage is:

$$\begin{aligned} &T_{FED} = \\ &(2W(2+2+3) + \frac{WH}{n}) + \\ &(\frac{2W}{2}(3+4+5+6) + \frac{WH}{4n}) = \\ &W(32 + \frac{5H}{4n}) \end{aligned} \tag{4.7}$$

Summing up the required clock cycles for one frame and dividing by the frequency, the total delay of our design is:

$$\begin{aligned} T_{delay} = \frac{1}{frequency} * (\frac{1.25WH}{n} + 32W) = \\ \frac{W}{frequency} * (\frac{1.25H}{n} + 32) \end{aligned} \tag{4.8}$$

The important difference in our work is the parameter n. If we use n=1, the throughput of our design is similar to that of Kalms' work [56] and the frame rate would be 98 frames per second. If we use n=4, which means having 4 memory sections, we can achieve 360 frames per second for the same image resolution (1024x768) at a maximum clock frequency of 102.7MHz (rounded off to 100MHz in Table 4.2 for ease of comparison with other work) on the Xilinx®Kintex®Ultrascale™FPGA. This number is also confirmed by our simulation results. We can readily synthesize this design for different image resolutions for various applications.

4.4 Experimental Results

In this section, we provide the implementation results and evaluation metrics of our work and compare our results with other related work. We use the KCU105 FPGA board which contains a Xilinx® Kintex® Ultrascale™ FPGA for synthesizing our design. Results demonstrate the performance of hardware design which is synthesized and simulated using Vivado® software.

Table 4.1 shows the resource usage of the stages of the design. In this table, LUTs are the Look up tables which are the smallest logic blocks in the FPGA. DSP represents the number of Digital Signal Processors which are the arithmetic units in the FPGAs and FF shows the number

of Flip Flops which represents the number of registers used in the design. Fig. 16 shows the power consumption of different stages of the design. The design consumes a total power of 1095mW. Table 4.2 demonstrates the overall resource usage, frequency and speed of our implementation in

Table 4.1: Resource consumption of the stages of the algorithm

Algorithm stages	LUTs	Block RAMs	DSP	FF
Diffusivity stage	22935	0	0	15016
FED stage	79454	0	29	43714
Preprocess stage	9187	0	0	5378
Memory management unit	620	524	2	805

comparison with other work. In comparison with the work by Jiang et al. [54] our work achieves higher frame rate, even though their work does not contain the contrast factor calculation. Our frame rate is higher than that of Kalms et al. [80], while our frame size is bigger. In comparison with Li et al. [44], our resolution is higher than their work, and still we use less LUTs (but more BRAM). If we use the same resolution as their work which is 640x480, our frame rate is 862 frames per second. Based on the results of Li et al. [44], their method affects the final accuracy. Therefore, with the same image resolution, our design achieves the highest frame rate using the same frequency.

Table 4.2: Comparison of design metrics

FPGA resources	Ours	Kalms et al.[56]	Jiang et al.[54]	Li et al.[44]
FPGA/Platform	Kintex Ultrascale	Zynq	ASIC	Kintex-7
LUT	112596	16507	—	196134
LUTRAM	72276	—	—	28068
BRAM	524	60	—	291
DSP	31	149	—	228
FF	65028	22738	—	157122
Image resolution	1280x720	1024x768	1920x1080	640x480
Frequency	100 MHz	100 MHz	200 MHz	100 MHz
Frame Rate	304 fps	98 fps	127 fps	784 fps

We designed and synthesized the proposed hardware using VHDL in Vivado® 2017 software. We also created a software model of the hardware in VHDL in MATLAB for accuracy evaluation purposes. This software model produces identical results as the hardware implementation. Since the focus of this paper is on non-linear scale-space generation, we do not need a complete matching system to compare the results. However, by adding the same key-point detector to both software implementation and the model of our hardware, we can use the repeatability metric to evaluate our design.

Other work has used different metrics to demonstrate the performance of their design. Jiang et al. [54] introduce a descriptor and report the performance of the whole system on the Oxford affine covariant features dataset [34]. Li et al. [44] use a self-synthesized dataset for accuracy evaluation. Kalms et al. [56] use FREAK descriptor and report the performance of the whole system which is

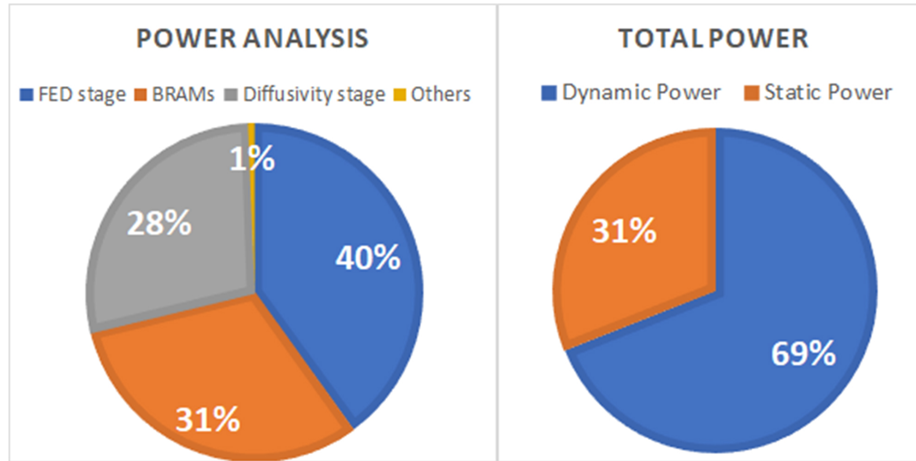

Figure 4.15: Power consumption. The left diagram shows the portion of power consumed by different stages of the algorithm. The right diagram shows the dynamic and static power consumption. Total power consumption of the design is 1095mW

also affected by FREAK descriptor. Since these work do not use the same metric for evaluation and the focus of our work is on non-linear scale-space generation, we decided to use repeatability [34] to show the correctness of the design. Higher repeatability implies improved performance of the feature detector which is the step after non-linear scale-space generation in an image matching system. Hence, this is an appropriate metric for demonstrating the performance of this design. This metric demonstrates how many of key points in the first image are found in the second image and is defined in equation 4.9:

$$Repeatability = \frac{\#of\ correspondences}{\#of\ key\ points\ in\ the\ first\ image} \tag{4.9}$$

We use the Oxford covariant features dataset [34] for comparing the repeatability of the software and the hardware implementation of the AKAZE algorithm. We use MATLAB® for software implementation of the algorithm. The Oxford affine covariant features dataset [34] contains a variety of image sets with different transformations such as changes in rotation, scale, viewpoint, and illumination. Each set has 6 images from which the results of matching key points of the first image with other images, are used in the evaluation. We add a Hessian detector to the non-linear scale-space images to find the key points for evaluation. The software implementation is based on floating-point and the hardware implementation uses integer arithmetic which is scaled to improve the computations. As shown in Fig. 4.16, the repeatability of the hardware implementation is close to the software implementation. The small difference is due to the approximations in bit-width in hardware design. We observe that for some of images, software is better and in other images hardware can be better. Since we are focusing on the non-linear scale-space filtering, approximations in bit-width have a direct effect on the output images. It may cut off some of the details from the images in lower bits. This could result in more matches in some images depending on the image content.

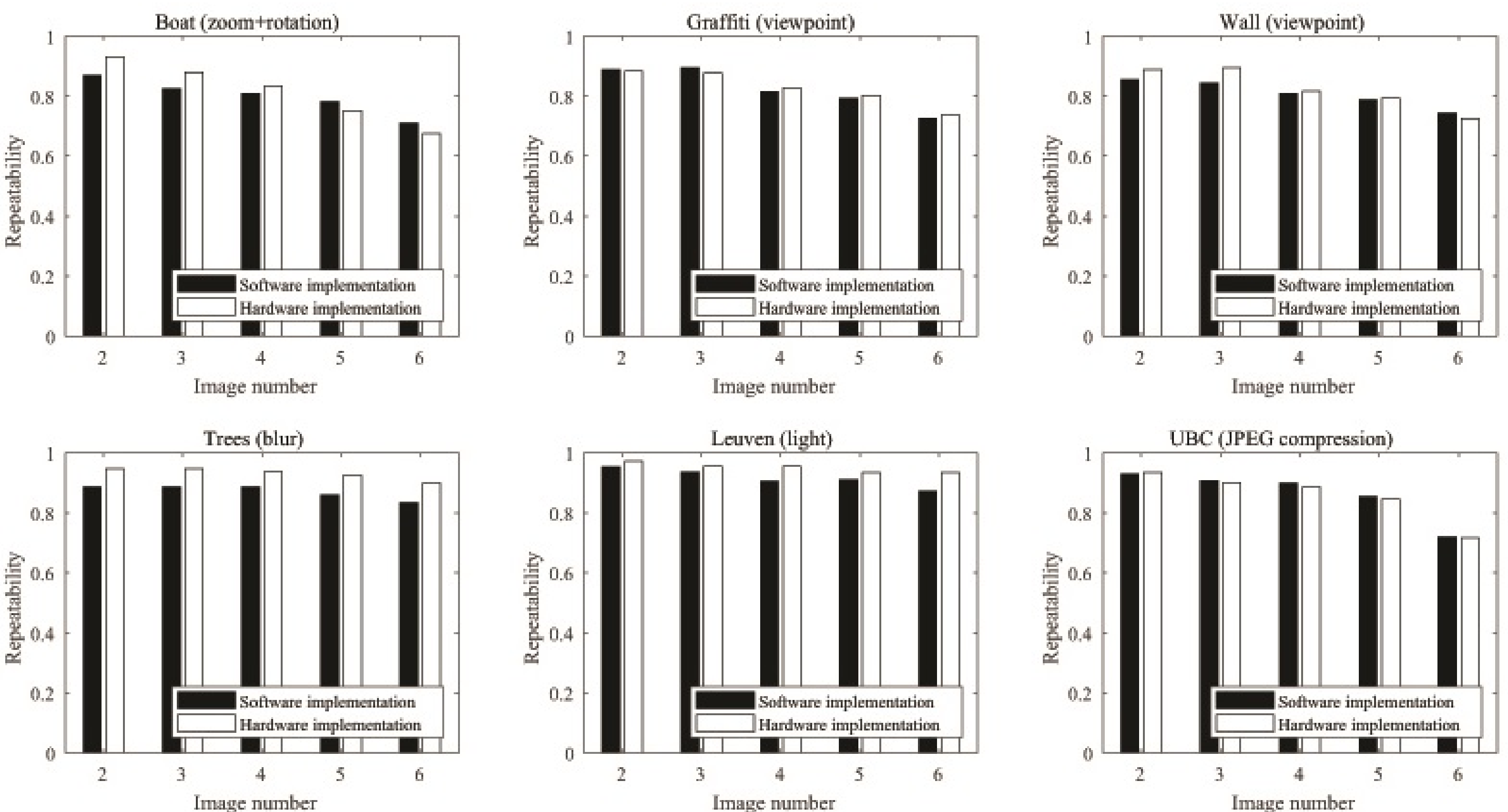

Figure 4.16: Comparison of repeatability between the software implementation and the hardware implementation based on simulation using image sets of the Oxford affine covariant features dataset [34]

4.5 Conclusion

In this chapter, we propose a design for non-linear scale-space generation for the AKAZE algorithm. Using non-linear scale-space for image matching leads to a higher accuracy but requires more computations.

The first contribution of this work is based on the idea to take advantage of the nature of the AKAZE algorithm which uses two passes through the image. This gives us an opportunity to use four parallel channels to generate a non-linear scale-space. In previous implementations of the AKAZE algorithm [56], the image data are read from an external memory in the first step to filter the image and compute the contrast factor. Then, the result is written back to the memory so that it can be read again for the next stage. We take advantage of this fact that in the first step, the image is read once from the external memory and we can have access to different sections of the image if we store it on chip in separate memories. Therefore, we design the memory management unit to store the image in 4 separate BRAMs so that we can generate the sub-levels of each section of the image in parallel. This, in addition to the fully pipelined architecture of each stage of the algorithm, leads to a noticeable speed up in our design.

The second contribution of this work is the architecture we propose for the second octave line buffers which uses the same data path as the first octave, but in a different scale. For this part, we introduce multi-scale line buffers which have several output windows for parallelizing the image input at different scales. Using traditional architecture results in consuming twice the number of the line buffer registers because each scale requires its own line buffers. However, by changing the architecture of the line buffers, we use the same hardware resources for both scales.

The third contribution of this work is the time-sharing mechanism in the memory management unit which provides the opportunity to process different sections of the image in parallel without having artifacts in the image. We introduce the time-sharing mechanism for this stage which has three phases in sections 4.2 and 4.3. By using this architecture, we can process multiple sections of the image which are stored in different memories in parallel and provide the border pixel values to all processing channels to prevent artifact in the images. With these contributions, we achieve 304 frames per second for 1280x768 image resolution. We demonstrate that the approximations proposed in our hardware implementation do not have a significant negative impact on the repeatability of the algorithm based on the results in Fig. 4.16.

Possible future avenues of investigation could include considering other diffusion algorithms to assess their suitability for hardware implementation and considering different detectors and descriptors that can be added to the current architecture, following the parallel channel processing concept.

Chapter 5

A Circular Shifting Binary Descriptor for Efficient Rotation Invariant Image Matching

In this chapter, our focus is on the orientation estimation and compensation in the patch description and key-point matching steps. The content of this chapter is based largely on our publication in the *International Conference on Pattern Recognition* [23]. We propose a method for enhancing the speed of patch description by removing the orientation estimation in the patch description step and compensating the rotation variation in the key-point matching step. In Section 5.1, we describe our proposed method in detail and in Section 5.2 we compare our results with other popular work using several experiments. Finally, we conclude this work in Section 5.3.

5.1 Circular Shifting Binary Descriptor

In this section, we explain our proposed method of circular shifting binary descriptors. The purpose is to generate the descriptor vectors in such a way that shifting the descriptors corresponds to the rotation of the sampling pattern. We propose to generate a sampling pattern so that each rotated version of the sampling pattern completely overlaps the non-rotated pattern.

We assume a group of m sampling points. Any two sampling points in a seed can be selected as a pair which will be used in a binary comparison test in the patch description step. Each seed has n_{pairs} pairs selected from 1 to n_{pairs_max}. The maximum number of pairs in a seed, n_{pairs_max} is calculated from (5.1):

$$\begin{aligned} & 1 \leq n_{pairs} \leq n_{pairs_max} \\ & \text{where } n_{pairs_max} = \binom{m}{2} = \frac{m!}{(m-2)!2!} \end{aligned} \tag{5.1}$$

For generating a symmetric sampling pattern, we rotate the seed $n_{rotations}$ times over 360 degrees.

If the seed covers θ_{seed} degrees, the number of rotations is calculated from (5.2):

$$n_{rotations} = \frac{360}{\theta_{seed}} \tag{5.2}$$

The total number of pairs in a sampling pattern is therefore:

$$n_{pairs_total} = n_{rotations} \times n_{pairs} \tag{5.3}$$

An arbitrary example of a sampling pattern seed is shown in Fig. 5.1.

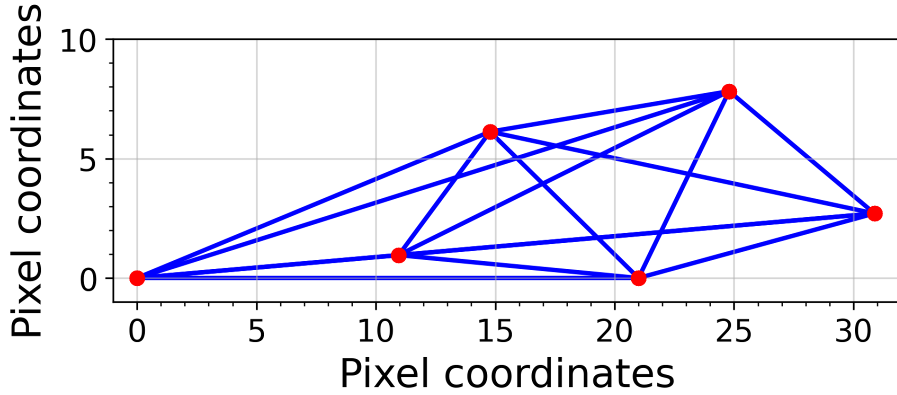

Figure 5.1: An example of sample pairs in a seed. In this seed, $m = 6$ samples points and $n_{pairs} = 15$ pairs are shown. The sample point on the origin (0,0) is the key-point.

The sampling pattern seed in Fig. 5.1 has $m = 6$ sampling points giving $n_{pairs} = 15$. In this example, the sampling pattern seed in Fig. 5.1 covers $\theta_{seed} = 22.5$ degrees of a complete circle and the coordinates of the sample points are selected arbitrarily. The numbers on the horizontal and vertical axes are the pixel coordinates and the key-point is at the origin (0,0). For larger or smaller patches, we can scale these coordinates.

By rotating and repeating this seed pattern $n_{rotations} = 16$ times from (5.2), a complete sampling pattern is generated for an image patch as shown in Fig. 5.2. The total number of pairs which is also the number of comparison bits, for this example $n_{pairs_total} = 240$, is calculated from (5.3). The sampling pattern in Fig. 5.2 demonstrates the sampling pattern created using the sampling seed shown in Fig. 5.1.

Figure 5.3 illustrates an example of rotating a sector of the sampling pattern (and pairs for comparison tests) for +45 degrees and their corresponding descriptors. In this example, each sector covers $\theta_{seed} = 22.5$ degrees and has 15 pairs, and since we have 16 rotations the descriptor vector has 240 bits represented by b_1 to b_{240}.

In order to get more accurate and robust orientation, smaller rotation is required. Based on our experiments, we select $\theta_{seed} = 5$ to increase accuracy. The value of $\theta_{seed} = 5$ results in $n_{rotations} = 72$ using (5.2). The number of pairs in the sample seed of Fig. 5.1 is 15, which results in 1080 bits (based on (5.3)). In order to examine the effect of using a smaller number of bits on the accuracy, we also performed experiments using $n_{pairs} = 3$ and $n_{pairs} = 7$ which result in 216 and 504 bits,

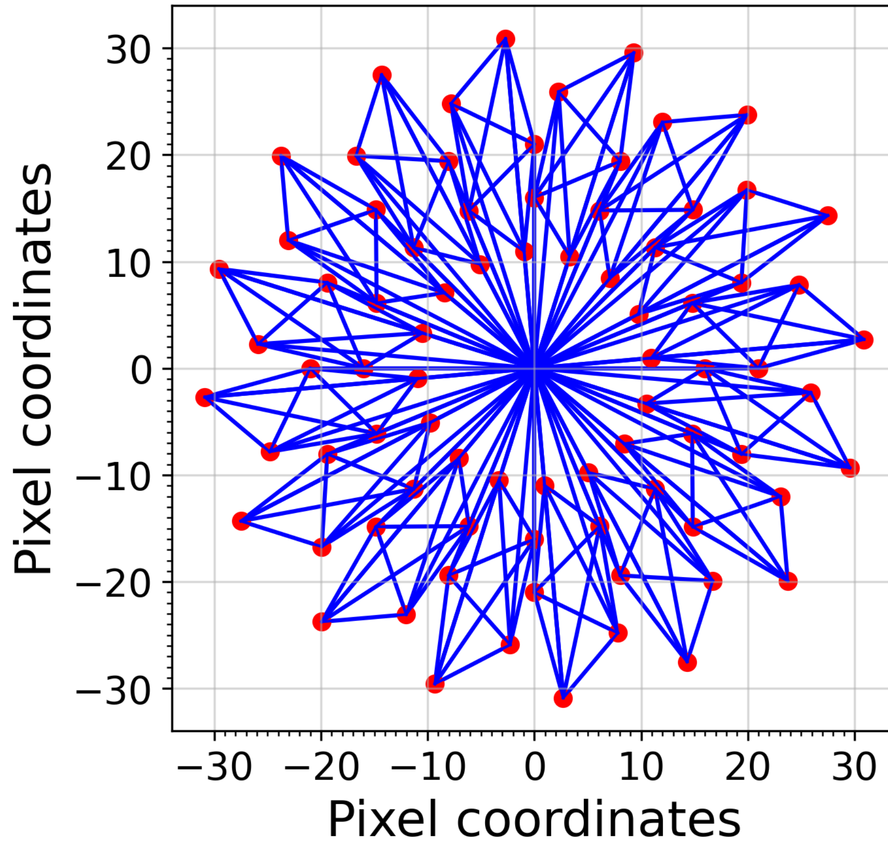

Figure 5.2: An example of sample points generated by rotating the sample seed $num_{rotations} = 16$ times. Each sample seed covers $\theta_{seed} = 22.5$ degrees.

respectively. The number of bits for each n_{pairs} is calculated from (5.3).

Three different configurations of sampling seed pairs and their final sampling patterns are demonstrated in Fig. 5.4. In this figure, the left pattern is an example of having only 3 pairs in the seed, the middle one is an example of 7 pairs and the right pattern uses all possible 15 pairs.

The descriptor vectors of the raw patches (without orientation compensation) are calculated for all the patches from the first and second images. An example of our proposed method is shown in Fig. 5.5. In this example, each sector has 5 comparison tests which result in 5 bits in the descriptor vector. As shown in Fig. 5.5, in the matching step, the descriptor of the first image is circularly rotated (72 times) and compared to the descriptor from the second image. The comparison is done by computing the Hamming distance which is efficient for binary descriptors.

Figure 5.6 shows an example of how the descriptor vector from a patch in the first image is circularly rotated. In this example, the descriptor vector has 5 bits from each sector and is rotated $n_{rotations}$ times. Empirically we chose $n_{rotations} = 72$ and as a result, each rotation corresponds to 5 degrees as in (5.2).

Prior rotation invariant descriptor algorithms in the literature require multiple operations for orientation estimation and compensation. The advantage of our method is that we do not need to

22.5º rotation

22.5º rotation

22.5º rotation
15 bit shift

$$descriptor\ vector = \{b_1, b_2, b_3, b_4, \ldots, b_{238}, b_{239}, b_{240}\}$$

22.5º rotation
15 bit shift

$$descriptor\ vector = \{b_{16}, b_{17}, b_{18}, b_{19}, \ldots, b_{12}, b_{13}, b_{14}, b_{15}\}$$

$$descriptor\ vector = \{b_{31}, b_{32}, b_{33}, b_{34}, \ldots, b_{27}, b_{28}, b_{29}, b_{30}\}$$

Figure 5.3: An example of rotating a sector of the sampling pattern (top) and its effect on shifting the descriptors (bottom) in our proposed method. Symbols b_1 to b_{240} represent 240 bits of the descriptor.

Figure 5.4: Examples of sample seeds (top) and the complete sample pairs generated (bottom). The left images represents 3 pairs. The middle images represents 7 pairs and the right images represents 15 pairs.

compute the orientation of the patch in the description step. Our proposed method has specific advantages in hardware implementation as we can compute the circular rotated versions of the descriptor and the Hamming distances in parallel. As a result, the descriptor generation in our method is faster than prior methods in the literature as we demonstrate in section IV.

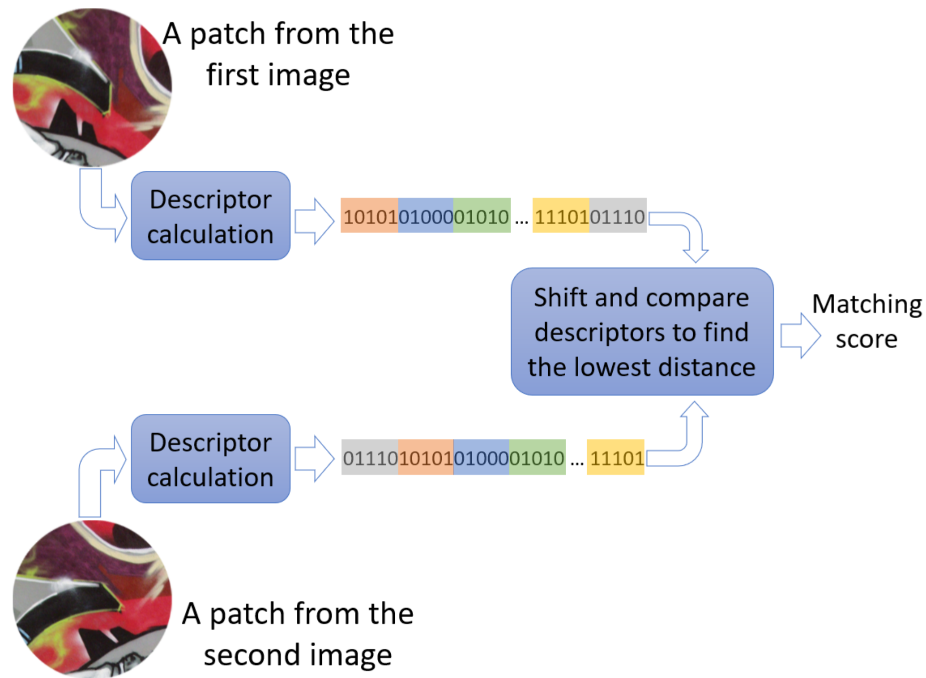

Figure 5.5: An example of our proposed method for two patches from the same key-point with different orientations from the first and second images.

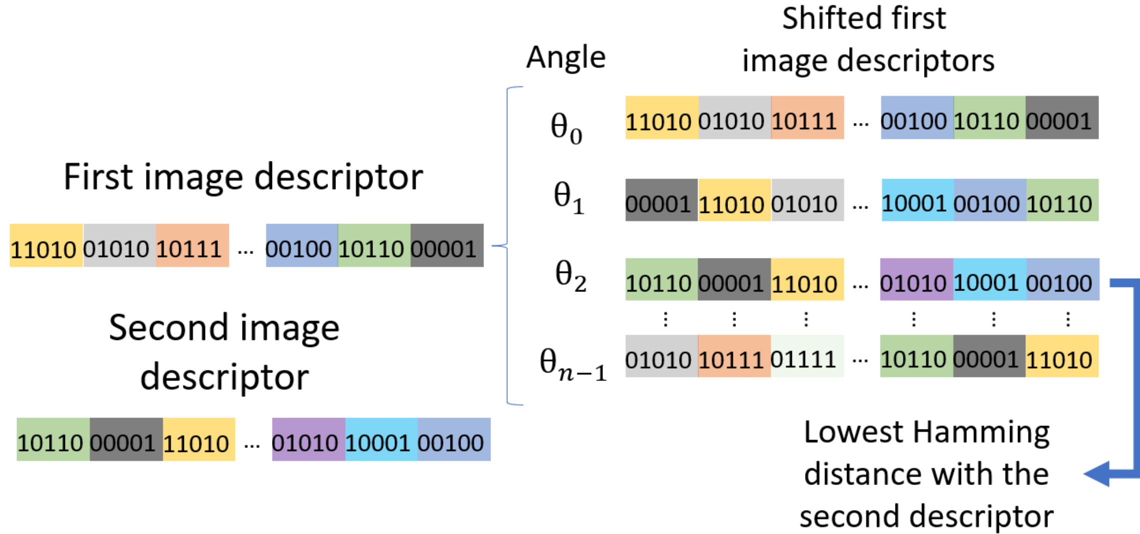

Figure 5.6: An example of circular rotation of the descriptor vector in our proposed method for two descriptor vectors from the same key-point with different orientations.

5.2 Experiments

In this section, we experimentally compare our proposed method with the ORB and BRISK algorithms which use a binary descriptor and the SURF algorithm which uses a non-binary descriptor. We use the implementations of ORB, BRISK, and SURF algorithms from the OpenCV library [81].

5.2.1 Number of Operations

Table 5.1 compares the number of operations required for orientation compensation in our method versus the ORB algorithm. The other steps of the image matching process such as key-point detection and patch description are assumed to be similar. In this table, the size of the image patch is W=65×65 which is a commonly-used patch size in the literature [82]. As an example, we set the number of key-points in the first image to $n_1 = 100$ and the number of key-points in the second image to $n_2 = 100$. We also assume $n_{rotations} = 72$ rotations for our circular shift method. Table 5.1 shows that our method does not require any of the operations for orientation estimation in the description step. Our method requires shifting of the descriptors and more Hamming distance calculations. However, circular shifting and Hamming distance operations can be implemented in parallel on an FPGA hardware platform for faster computation.

Table 5.1: Comparison of the number of operations required for orientation compensation

Operations per two images	ORB orientation estimation	Our proposed method
Multiplication	$2 \times (W/2)^2 \times \pi \times (n_1 + n_2) = 5306600$	0
Summation	$2 \times (W/2)^2 \times \pi \times (n_1 + n_2) = 5306600$	0
Division	$n_1 + n_2 = 200$	0
Arctangent	$n_1 + n_2 = 200$	0
Hamming distance	$n_1 \times n_2 = 100000$	$n_1 \times n_2 \times n_{rotations} = 7200000$
Shift operation	0	$n_1 \times n_{rotations} = 7200$

n_1: The number of key-points in the first image
n_2: The number of key-points in the second image
$n_{rotations}$: The Number of rotations in our method
W: Size of the image patch

5.2.2 Comparison of Rotation Error

In this experiment, we first extract 100 key-points using the ORB detector from each image in the Hpatches dataset [82]. Then, we rotate the patch around each key-point for 360 degrees in steps of 1 degree. In each rotation, we compute the descriptor. For SURF and ORB descriptors, we compute the main orientation of the rotated patch. For our proposed method, we select the orientation of the circularly rotated descriptor which has the least Hamming distance with the unrotated patch. In the next step, we compute the error from the rotation value in each degree of rotation. Finally, the average rotation error in each degree is computed over all images in the dataset. The pseudocode of our experiment is shown in Algorithm 2.

Figure 5.7 shows the mean rotation error of ORB, BRISK, SURF compared with our proposed algorithm using 216 bits. Table 5.2 presents the mean absolute and standard deviation of the error values displayed in Fig. 5.7. The ORB algorithm has the least mean absolute error which is close to that of our proposed method. We only report the rotation error for our method using 216 bits since the rotation errors using 216 bits, 504 bits, or 1080 bits configuration are close to each other. However, the number of bits has a direct effect on the overall accuracy of the image matching algorithm. Overall, the results in this section shows the performance of our proposed method in handling the rotation of the image patches.

Algorithm 2 Pseudocode for the rotation error experiment

```
Input : Images from Hpatches dataset
Output: Error_vector
Key − points ← Extract 100 key-points from each image
               using ORB detector
Patches ← local neighborhoods of 65 × 65 pixels around
          each key-point in Key-points
N ← Total number of Patches
Error_matrix ← A matrix of N rows and 360 columns
for i = 1 to N do
    if descriptor is SURF, ORB, or BRISK then
        O1 ← Compute the orientation of Patches(i)
        ; O1 is the initial orientation
    else if descriptor is our method then
        O1 ← 0
    for θ = 0 to 359 do
        Patch_rot ← Rotate Patches(i) for θ degrees
        if descriptor is SURF, ORB, or BRISK then
            O2(θ) ← Compute the orientation of Patch_rot
        else if descriptor is our method then
            O2(θ) ← Select the orientation of Patch_rot that
                    leads to the minimum Hamming
                    distance with Patches(i) based on the
                    circularly shifted descriptor
        Error_matrix(i, θ) ← O2(θ) − O1 − θ
for θ = 0 to 359 do
    Error_vector(θ) ← Compute average of errors for
                      each column in Error_matrix
```

Table 5.2: Comparison of descriptors in rotation error on HPatches dataset

Algorithm	mean Absolute Error	Standard deviation
SURF	6.81	7.35
ORB	0.55	0.63
BRISK	1.40	1.53
Ours 216 bits	0.70	0.65

5.2.3 Comparison of Image Matching Accuracy

We evaluate the new sampling pattern in terms of mean Average Precision (mAP) on the Hpatches dataset [82]. The mAP is a commonly used metric for evaluation of image matching algorithms and is computed as the area under the precision-recall curve [82].

Figure 5.8 presents an example of mAP in four of the image sets in the HPatches dataset. In each graph, the average precision of matching the points from the first image to the other images are displayed. The last column of each graph is the mAP of that image set. This figure shows that in some cases the ORB descriptor is better than our proposed method while in other cases our method outperforms ORB. Overall, our results are comparable with the ORB descriptor in terms of accuracy. However, our method does not require the orientation computation step.

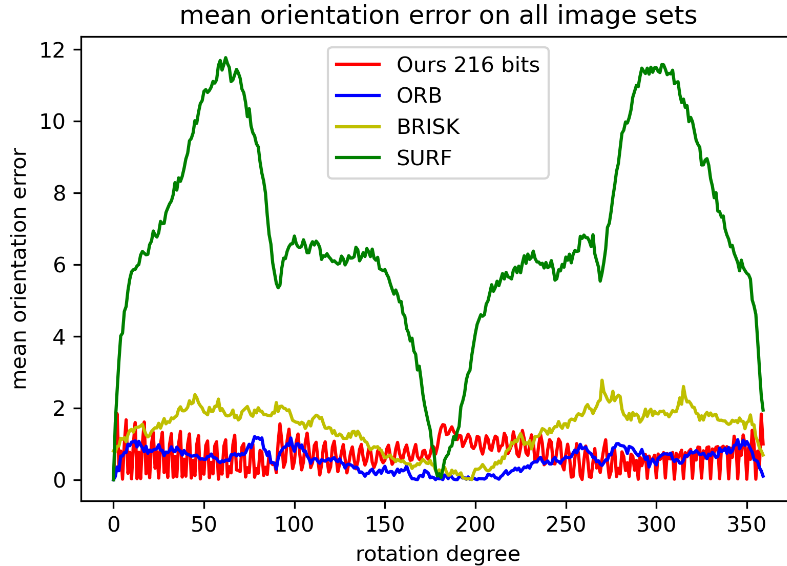

Figure 5.7: Comparison of rotation errors of our proposed method with ORB, BRISK, and SURF on the HPatches dataset [82].

Figure 5.9 illustrates the mAP metric of our proposed method using various number of bits in comparison with the ORB algorithm over the whole HPatches dataset. The mAP of our method increases as the number of bits increases since using a larger number of bits we can extract more information from an image patch. Based on our experiments, we get comparable results to the mAP reported in Fig. 5.9 by having small changes on the location of the points in the sampling pattern. In future, we will optimize the location of the sampling patterns to maximize accuracy. The results show that our method is comparable in accuracy to the ORB algorithm. Our method requires a smaller number of computations than ORB as shown in section IV.A. It is also important to note that ORB uses a learning method to select high-variance, low-correlated pairs to select the best pairs while we have selected the sampling pairs arbitrarily.

5.3 Conclusion

In this work, we proposed a method for enhancing the speed of calculation of rotation invariant binary descriptors. The sampling pattern is generated using a sample seed which is rotated around each key-point. Instead of computing the orientation in the description step, the generated descriptor is circularly rotated in the matching step so that the best rotation for matching is selected. It was demonstrated that the number of operations in our proposed method is less than the other methods while the rotation error of our method is similar to other methods and the accuracy of image matching is comparable.

In future experiments, we will focus on increasing the accuracy of our model by tuning the location of the sampling patterns. In addition, our method can be implemented in parallel for attaining

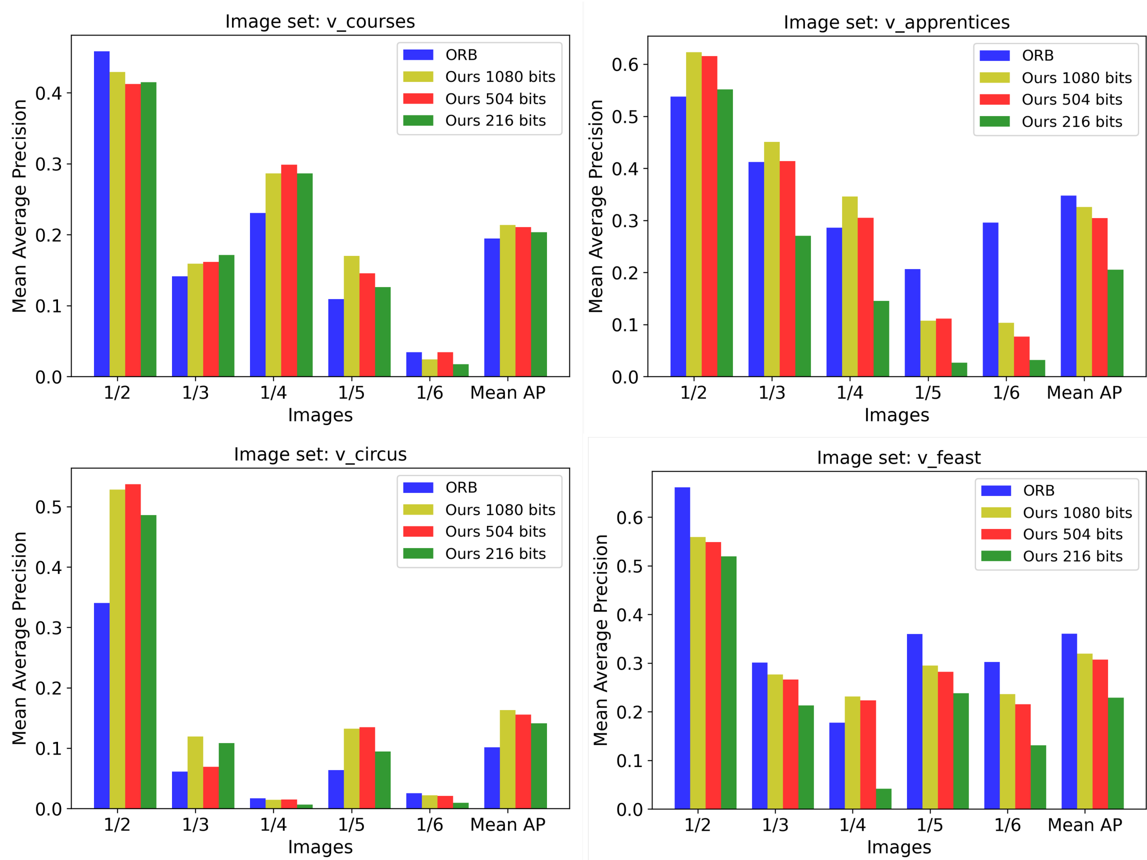

Figure 5.8: Examples comparing the mAP of our proposed method with the ORB algorithm.

higher speeds in hardware. We will examine hardware architectures for efficient implementation of the circular shifting binary descriptor.

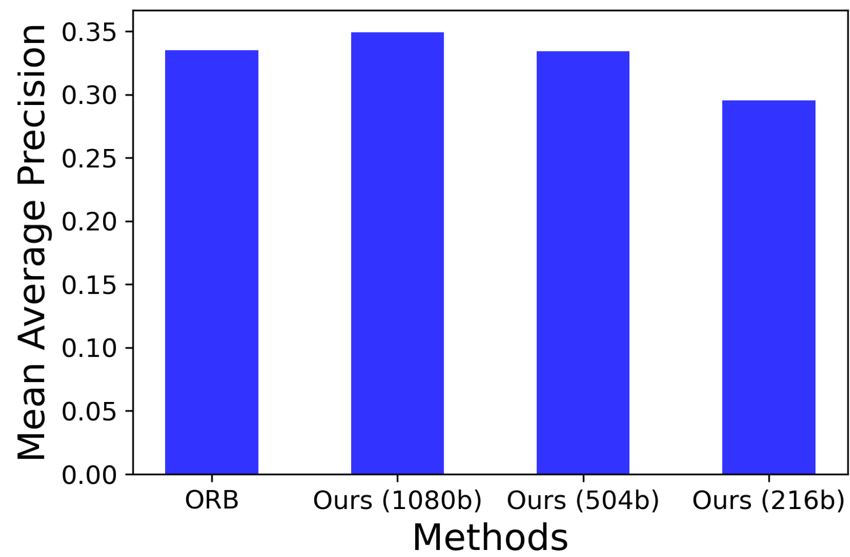

Figure 5.9: Comparison of our method and the ORB descriptor over the HPatches dataset [82].

Chapter 6

A Modified CAM Architecture for Improving Binary Descriptor Matching

In image matching, reference image and target image might be captured from different view points and illuminations which might cause differences in the binary descriptors of patches around key-points of the same local points. In binary descriptor matching, if the Hamming distance of a descriptor from the reference image and a descriptor from the target image is within a pre-define threshold, the corresponding key-points to those descriptors are proposed as a match. Therefore, in the binary descriptor matching step of the image matching algorithm a descriptor in the reference image which is an approximate value of the query descriptor (descriptor corresponding to the key-point of the target image) is acceptable as a match.

Although CAM can locate a query data in memory with high speed, it is not possible to use CAM for search applications such as binary descriptor matching in which approximation is also acceptable. Riazi et al. [83] illustrate that CAM cannot be used directly for Hamming distance calculation. The distance between two separate binary descriptors is based on the difference between their bits while in CAM, a match is found only if the query descriptor is an exact match to at least one of the contents of memory. Therefore, conventional CAM architecture cannot be utilized for binary descriptor matching due to the lack of capability to tolerate approximation. Our CAM-based approach supports approximate matching and provides the advantage of finding the matching descriptor to a set of binary descriptors much faster than other methods. This is beneficial where a large number of Hamming distance calculations are required, including real-time image matching [84], or applications that require processing of large databases such as memory-augmented neural networks [85] and DNA sequencing [86].

In this chapter, we modified the partitioned CAM architecture to use in binary descriptor matching to increase the speed. In our method, for each query binary descriptor of target image, the nearest binary descriptor of reference the image is proposed using a modified partitioned CAM, and only the Hamming distance for the query and proposed binary descriptor is calculated. So for a single

query binary descriptor no iteration through all binary descriptors of reference image is required.

We present our modified CAM design for binary descriptor matching in Section 6.1. We report timing analysis and resource utilization of our implementation of our model in Section 6.2. We conclude in Section 6.3. The content of this chapter has been submitted for publication.

6.1 A New Architecture for Binary Descriptor Matching Based on CAM

In this section, first, we present our design to use CAM for binary descriptor matching and illustrate with an example in Section 6.1.1. Then, we discuss the effect of selecting the number of bits for CAM partitioning on binary descriptor matching in Section 6.1.2. Finally, we discuss the process timing of our method in Section 6.1.3.

6.1.1 A Novel Modified Partitioned CAM for Binary Descriptor Matching

In this section, we present our modifications on the partitioned CAM [71] so that it can be used for binary descriptor matching. We modify the partitioned CAM design in order to load the location of the closest data (approximate data) instead of just loading the exact data. Although we use the idea of partitioning CAM modules for more efficient implementation, the novelty and contribution of our work is introducing a method for using CAM for descriptor matching. CAM partitioning is a method for more efficient implementation. However, none of the variants of CAM implementation can be used directly for descriptor matching since they do not have tolerance for accepting a small number of bit differences between the two descriptor vectors. The additional logic elements and the data processing methodology which enables the use of a high performance CAM for descriptor matching is a primary contribution of our work.

The pseudocode for our approach is shown in Algorithm 3. The input is query data, which has N_b bits, and it is represented by a bit string ($b_{(N_b)-1}$ to b_0) and is divided into k strings of m bits (k is the number of CAM units). Each m-bit string is the input of a CAM unit and the corresponding content loaded from each CAM unit is an N-bit binary string. The loaded contents from CAM units are represented as a matrix $A_{k\times N}$, where each row corresponds to the output of one CAM unit and each column corresponds to a location of the data. In the subsequent step, the summation of each column of A is calculated and stored in the Sum_values matrix. By doing so, the summation result of each column ($Sum_values(i)$) shows the number of CAM unit outputs with value 1 as the output bit for the corresponding index (index i). After calculating the summation, the index of Sum_values, which contains the maximum value among the elements of Sum_values string, represents the location of the closest content to the query data (b_{N_b-1} to b_0). If there are two maximum values, the left most position is selected. To select the contents within a pre-defined distance, we compare the maximum value with a pre-defined threshold ($Sum_val_threshold$), which is determined experimentally in section 6.2.3. We then select the content as the output (the closest content to the query data) if the maximum value is greater than or equal to the $Sum_val_threshold$. The output of this algorithm is the location of the approximate value of the query data.

Algorithm 3 Pseudocode for our approach to search for an approximate value of the query data using a modified partitioned CAM.

```
Input: Query data (b_{N_b-1} to b_0)
Output: Location //Location of approximate value
Parameters:
    N_b = Number of bits for each descriptor
    k = Number of CAM units
    m = N_b/k Number of bits of the input of each CAM unit
for i = 1 to k do
    //Loading from CAM units (all in parallel)
    //A = A matrix of dimension k × N
    A_(i,1:N) = load from CAM_i[b_{i×m-1}...b_{(i-1)×m}]
for i = 0 to N-1 do
    //Implemented all in parallel
    //Counting the number of 1s in each column
    Sum_values(i) = compute summation of A_(1:k,i)
Max_val = select the maximum of Sum_values
if Max_val ≥ sum_val_threshold then
    Location = Index of max(Sum_values)
else
    //No data found.
```

Fig. 6.1 demonstrates an example of using partitioned CAM units to locate the exact match and our modification approach to partitioned CAM to locate an approximate match in a RAM. In this example, the query data is 101011, comprising 6 bits ($N_b = 6$), and three CAM units ($k = 3$) are used to find a match for the query data stored in RAM (as in Fig. 6.1 (c)) as shown in Fig. 6.1 (a). The result of processing the loaded values of CAM units for the partitioned CAM and our modified partitioned CAM is shown in Fig. 6.1 (b). If the loaded values were processed in a conventional partitioned CAM module, the result of the logical bit-wise AND of the bits has no non-zero bit as shown in Fig. 6.1 (b). So, the partitioned CAM cannot locate any content in the RAM (no hit). If the same loaded values of CAM units are processed using our approach for modified partitioned CAM, instead of bit-wise AND, the number of ones in each column of CAM unit outputs is counted using bit-wise addition. This addition result is compared with a pre-defined threshold and if it is greater than or equal to the threshold, the index is selected as the location of the approximate match. As shown in Fig. 6.1 (c), the content of address 4 in RAM has only one bit difference with the query data. As the identical value 101011 is not stored in the RAM, the conventional partitioned CAM cannot find a match for the input query, while our approach finds the closest value (approximate value) and proposes location 4 in the RAM as the output.

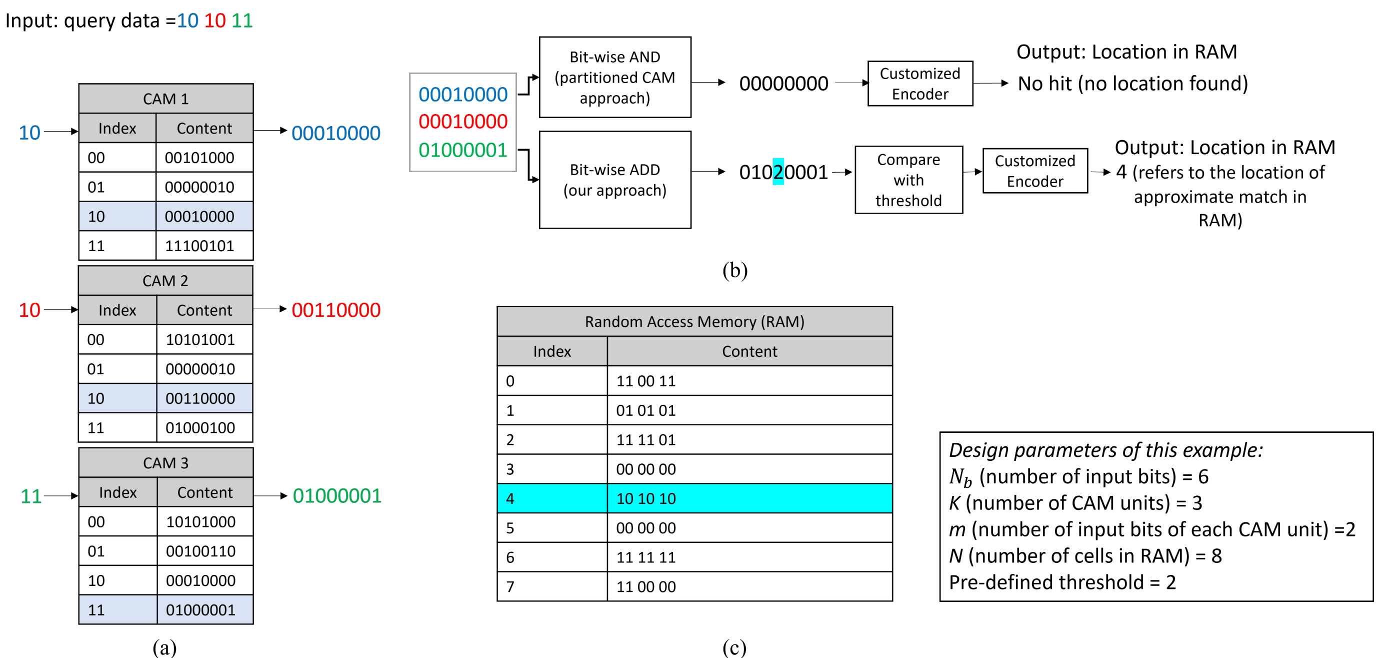

Figure 6.1: An example of finding a match in a RAM using partitioned CAM units for conventional partitioned CAM and our approach for modified partitioned CAM. (a) Shows the query data and three CAM units, the selected content based on the input index is highlighted for each CAM unit. (b) Shows the outputs of the conventional partitioned CAM and our modified partitioned CAM. (c) Shows the content of the corresponding RAM. The selected content in the RAM unit is highlighted.

The block diagram of our modified partitioned CAM for binary descriptor matching is shown in Fig. 6.2. The inputs of this design are a key-point and its corresponding descriptor of a target image. The goal is to match a key-point of the target image with one of the key-points of a reference image stored in a RAM (Reference Image Key-points RAM in Fig. 6.2). The outputs of the design are a key-point of reference image and a key-point of target image that are selected as a match. The matching of key-points is done based on the distance of their corresponding descriptors. The corresponding descriptor of the input key-point is thus the query descriptor of the modified partitioned CAM.

In this design, there are two RAMs to store descriptors and key-points of the reference image. The index of each key-point stored in the Reference Image Key-points RAM is the same as the index of its corresponding descriptor in the Reference Image Binary Descriptors RAM. Therefore, one index (named Location of approximate match) is used to load both the key-point and its corresponding descriptor. For each query descriptor (descriptor of target image), a descriptor among descriptors of the reference image (stored in the Reference Image Binary Descriptors RAM) is selected using our modified partitioned CAM. The Hamming distance of this selected descriptor and the query descriptor (target image descriptor) is calculated. If the distance is less than a pre-defined threshold ($Hamming_threshold$), which is determined experimentally based on *Precision* and *Recall* values as shown in section 6.2.3, the corresponding key-points are proposed as a match. The *Precision* metric and the *Recall* metric which are formulated and shown later in (6.2) and (6.3), determine how many of the proposed matches are correct and how many of the correct matches from the detected key-points are selected, respectively.

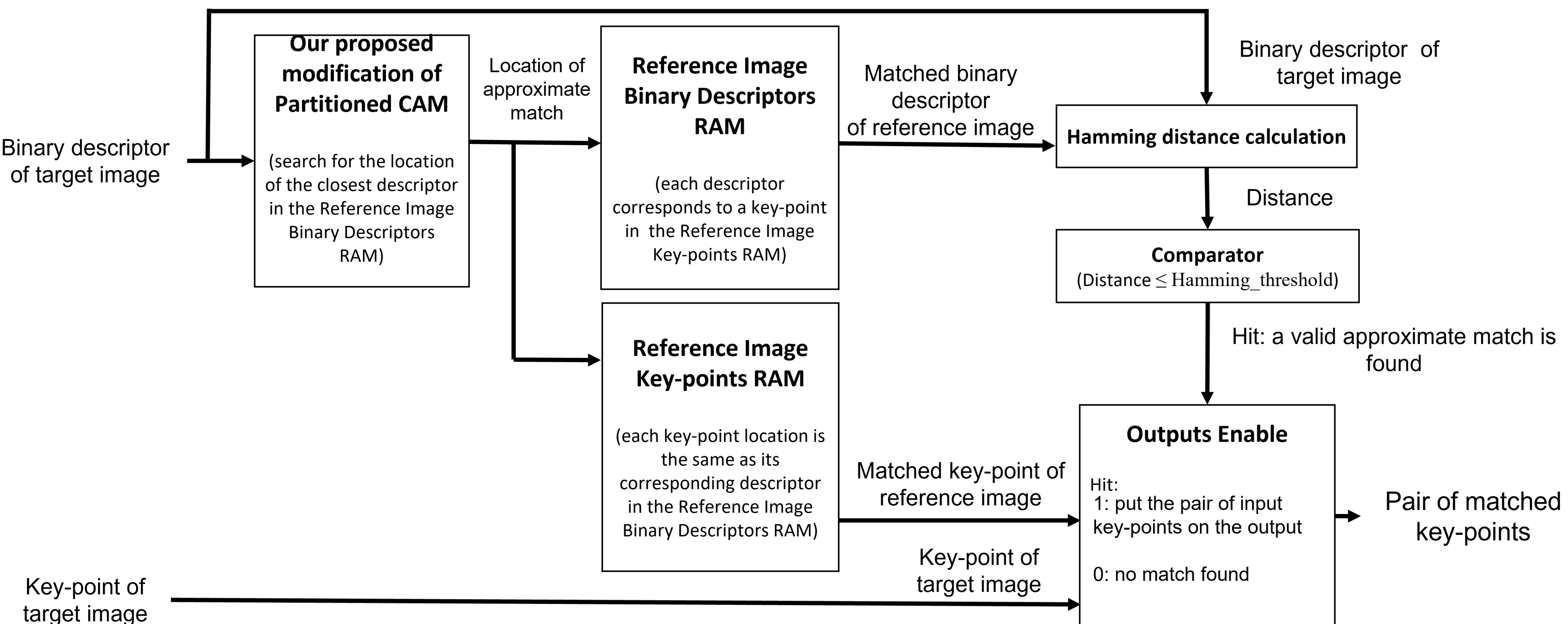

Figure 6.2: Block diagram of our modified partitioned CAM approach for binary descriptor matching. The inputs are a key-point of the target image with its corresponding descriptor. The outputs are a key-point in the target image and a key-point in the reference image that are matched based on their binary descriptors distance.

The data flow of our method for modified partitioned CAM implementation is shown in Fig. 6.3 (a). In order for CAM to support approximate matching (locate the closest value), we modify the partitioned CAM structure (shown in Fig. 2.8 (b)) by replacing the bit-wise AND with an adder tree to calculate the Sum result (Sum_values) and a comparator tree to select the maximum value (Max_val). By using an adder tree for each column, the summation of all bits is computed (counting the number of 1s). Then, the maximum of the summation values is selected using a comparator tree. Next, an N-bit binary vector showing the $Sum_values(i)$ with the Max_value is generated. Element i of the binary vector is $'1'$ if $Sum_values(i)$ is equal to the maximum value (Max_value); otherwise, element i of the binary vector is $'0'$. Finally, a priority encoder converts the binary vector to the output locations of the query data. Fig. 6.3 (b) demonstrates the structure of the comparator tree in Fig. 6.3 (a). The comparator tree consists of $\lceil log_2(N) \rceil$ layers where N is the number of key-points for each image. Each layer in Fig. 6.3 (b) contains a number of Comparator units (comparator and multiplexer) as shown in Fig. 6.3 (c). The Comparator unit compares two values from the previous layer, and by applying the *Greater than (Gt)* signal of the comparator as the select signal of a multiplexer, passes the greater value to the next level. The comparator tree structure selects the maximum value among all summation results.

The modified partitioned CAM architecture can locate the content closest to the query data without any iteration.

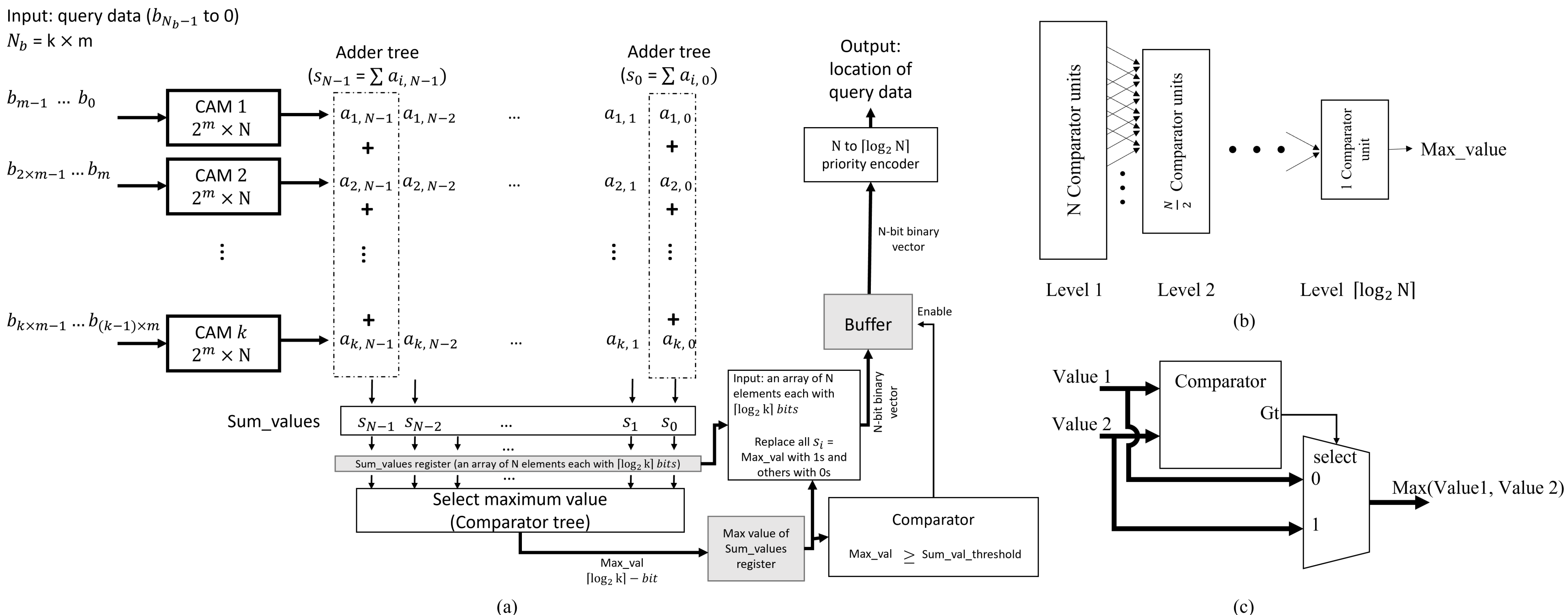

Figure 6.3: (a) The architecture of our matching module using partitioned CAM. The gray boxes are registers. The summation result is computed using N adder trees. The parameter m shows the number of input bits for each CAM unit, and k indicates the total number of CAM units. If the descriptor has 256 bits and $m = 4$, the number of CAM units is $k = 64$. (b) The comparator tree for selecting the maximum value using Comparator units shown in (c). N is the number of key-points in the reference image. (c) The architecture of each Comparator unit which outputs the maximum of its two inputs (Value 1 and Value 2).

6.1.2 Selection of the Number of Bits for Partitioning the Query Descriptor

In this design, the parameters that can be tuned are N_b and k, based on the trade-off made for binary descriptors matching performance (in terms of *Precision* and *Recall*) and resource utilization. As the number of bits for the binary descriptor (N_b) is a parameter that is usually tuned based on the performance of the previous step of the image matching algorithm (patch description), we can assume N_b is a fixed number. In Fig.6.3 (a), k is the total number of CAM units, and $m = \frac{N_b}{k}$ is the number of bits from the descriptor dedicated to the address of each CAM unit. For a fixed number of bits for a descriptor, changing the parameter k also changes m and affects the accuracy and resource utilization of the architecture. If m, which relates to the size of the CAM units increases, the matching system detects exact matches on longer bit strings of the descriptor vectors. This may result in an increase in matching error by missing the correct matches that have only a few different bits.

Fig. 6.4 shows an example of two 8-bit binary descriptors ($N_b = 8$). We assume the binary *descriptor* 1 is the query binary descriptor (target image binary descriptor), and binary *descriptor* 2 is the reference image binary descriptor which is stored in location i in the Reference Image Binary Descriptor RAM. The Hamming distance of these two descriptors is equal to 1 as the least significant bit of these binary descriptors is the only different bit in these binary descriptors. If we assume the pair of *descriptor* 1 and *descriptor* 2 is a match, in order for the design to select *descriptor* 1 as a match of *descriptor* 2, the ith bit of loaded outputs of the CAM units should be 1 for a pre-defined number of CAM units (refer to Fig. 6.3 (a) in which (S_i) should be greater than or equal to the pre-defined threshold ($Sum_val_threshold$)). If the ith bit of output for CAM unit i is equal to 1, we consider it as a hit for that CAM unit. Otherwise, we consider it a miss. As shown in Fig. 6.4 (a), for $k = 4$, there are three hits and one miss. In 6.4 (b) for $k = 2$, there is one hit and one miss. In 6.4 (c), there is 0 hit and one miss. In our design, after this stage, if the number of hits is greater than or equal to the pre-defined threshold ($sum_val_threshold$), we consider the pair of binary descriptors to be a match. Therefore, the probability of selecting *descriptor* 1 and *descriptor* 2 as a match is increased when the number of CAM units (k) is increased.

Consequently, increasing the number of CAM units k results in increasing the performance of image matching in terms of Precision while the resources required for the adder tree and comparator tree also increase.

Another important parameter in choosing the number of CAM units for partitioning the query binary descriptor is the total number of bits required for the CAM implementation. Each CAM unit requires $2^m \times N$ bits where m is the number of bits in each partition and N is the number of key-points. The total number of memory bits for the CAM units is shown in (6.1):

$$\text{Number of memory bits} = \frac{N_b}{m} \times 2^m \times N \tag{6.1}$$

where N_b is the total number of descriptor bits ($N_b = k \times m$ as in Fig. 6.3 (a)). Therefore, if $N_b = 256$, for $m = 2$, $m = 4$, and $m = 8$, the total number of required memory bits are $2^9 \times N$, $2^{10} \times N$, and $2^{13} \times N$, respectively. In this work, we chose $m = 4$, which results in 64 CAM units

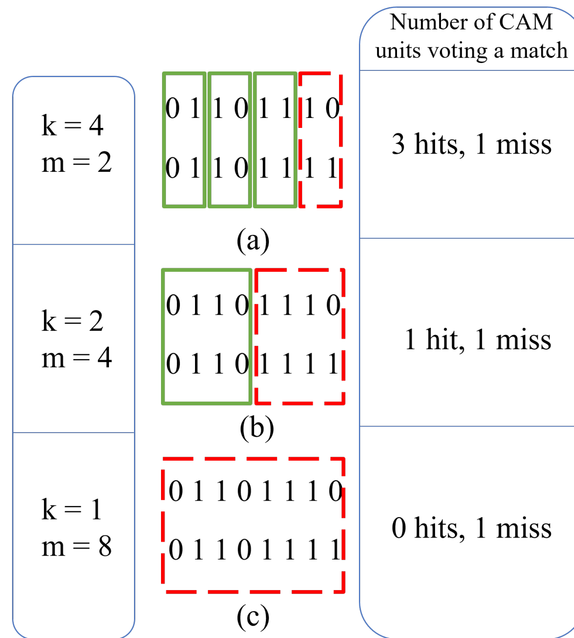

Figure 6.4: An example of a various number of CAM units and their effect on the number of hits for matching the two 8-bit binary descriptors with only one bit difference. Partitioning with a higher number of CAM units (for example, $k = 4$ in (a) results in a higher number of hits, while the lower number of CAM units (for example, $k = 1$ in (c)) leads to no hit.

after considering a trade-off between accuracy and resource utilization.

The approximation mentioned throughout this paper is not a source for error. In fact, by introducing this approximation (computing the summation of the bits from the CAM outputs and partitioning the bits of the descriptor for storing in the CAM modules) we create the possibility of using CAM for image matching which results in increased speed for the matching step. Our proposed method finds the best match from the first image to the second image. Descriptor matching by nature should accommodate small differences in the bit values. The only potential error that may happen is if we increase the size of the CAM modules (m) so that the matching system looks for an exact match of the descriptor from the first image to the second image. This may result in not finding an exact match and missing the correct matches that have a small number of unmatched bits.

6.1.3 Process Timing of the Proposed Modified Partitioned CAM

The modified partitioned CAM binary descriptor matching has three steps. In this section, we present illustrative timing diagrams to show the conceptual description of the state machine operation. The diagrams in this section visually illustrate how our proposed hardware architecture works in each step.

Step 1

Fig. 6.5 shows the timing diagram of the first step. In the first step, the corresponding locations to the descriptors of the reference image are written in CAM units. Storing each descriptor location in CAM requires three clock cycles in our design: the first clock cycle is for loading the existing value from each CAM module. The input of each CAM module is the number of selected bits (4 bits in our design) from the descriptor. In the second clock cycle, the corresponding bit of CAM that represents the descriptor location in the data memory is set to 1.

At last, the new value is stored in the same cell of CAM in the third clock cycle. The first step is processed immediately after the key-point detection and patch description steps.

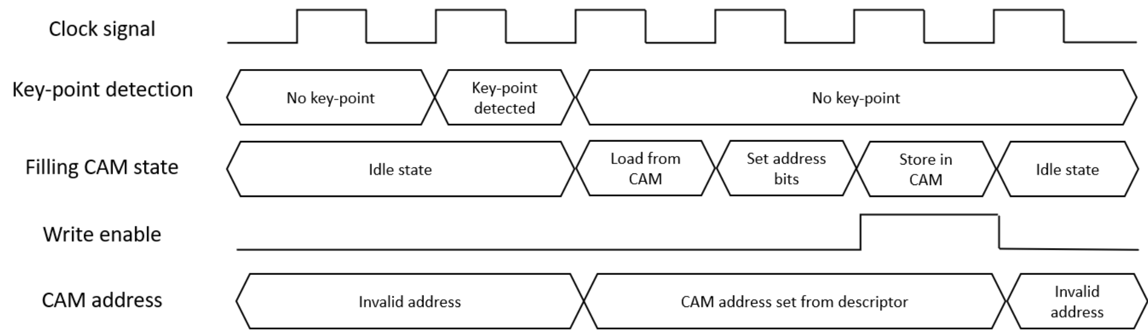

Figure 6.5: Timing diagram showing the filling of the CAM. After detecting a key-point, three clock cycles are required to fill the CAM with the location of the descriptor corresponding to the detected key-point.

Step 2

The second step of our CAM-based binary descriptor matching is finding the most similar descriptor from Reference Image Binary Descriptors RAM to the query descriptor (target image descriptor). First, we partition the query descriptor into m-bit values. Each m-bit array is used to access the data stored in one of the k CAM units. Then, the outputs of the CAM units are loaded in one clock cycle, and the summation of each column is calculated using an adder tree. The timing diagram of the second step is shown in Fig. 6.6. In Fig. 6.6, each t_n corresponds to one clock cycle, and at each clock cycle, four key-points are being processed in the pipeline structure.

Step 3

The third step is writing zeros in all memory cells to prepare the CAM for the next image. This step happens once for each image and is done in parallel for all CAM units. The required time for this step is equal to the number of cells in each CAM unit (in our design, the number of cells in each CAM is equal to 16 as the input of CAM has 4 bits). In comparison with the number of clock cycles required for processing each image, the number of clock cycles for this step, 16 clock cycles, is negligible.

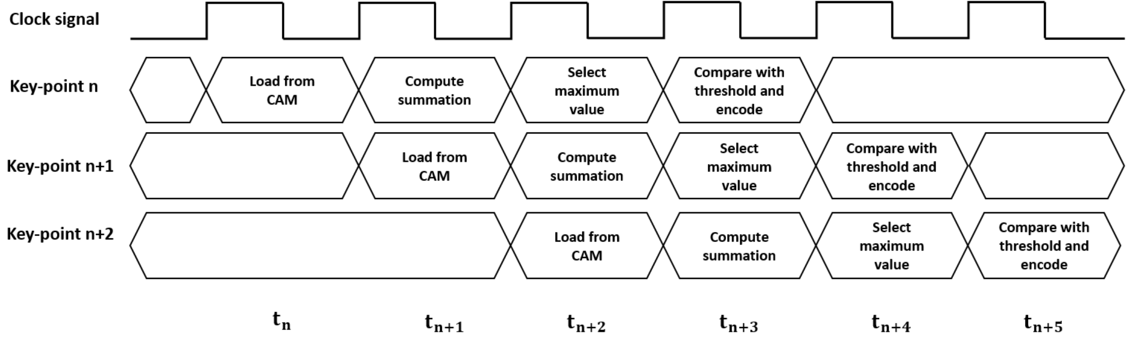

Figure 6.6: Timing diagram showing the reading from CAM for three consecutive key-points. At each clock cycle, the pipeline registers are filled with the key-point data.

6.2 Experimental Results

In this section, we present the result of the simulation and implementation of our binary descriptor matching using CAM implemented on a Kintex® UltraScale™ (KCU105) FPGA board [87] which contains a Xilinx® XCKU040-2FFVA1156E FPGA.

6.2.1 Time Complexity

The descriptors from the reference image are generated in sequence and stored in the CAM. In a conventional descriptor matching system, the descriptors from the reference image and the target image are stored in the memory. The reference image descriptors are loaded one by one to be compared with each descriptor from the target image. In our CAM-based circuit, although the descriptors from the reference image are stored in the CAM sequentially, the step which compares them one by one with the descriptor from the target image happens without any iterations due to the modified CAM architecture. This is possible due to the nature of the CAM which provides the address of the query data in the memory and our proposed approach which enables the CAM to accommodate bit differences as required in the descriptor matching problem. As a result, only a single pass based on the number of descriptors of the target image is required.

The commonly used method for binary descriptor matching is the computation of the Hamming distance for the two descriptors from the reference image and the target image. Let the number of key-points in the reference and target images be N_1 and N_2, respectively. The basic conventional commonly-used matching based on the Hamming distance computation method requires $N_1 \times N_2$ Hamming distance operations. Therefore, the time complexity for binary descriptor matching is $O(n^2)$ where n represents the number of arguments (in our case, the number of key-points) passed to the algorithm. Many hardware implementations of Hamming distance calculation such as [64] have proposed parallel Hamming distance modules, leading to a time complexity of $O(n^2/m)$, where m is the number of parallel Hamming distance calculation modules. Since m is a bounded value and is smaller than the number of key-points in practical applications, the time complexity of these methods is still $O(n^2)$.

In conclusion, our method features high parallelism due to the CAM architecture. Each binary

descriptor of the target image is compared with all binary descriptors from the reference image simultaneously (due to the nature of CAM functionality) without any one-to-one comparison. Therefore, the required time for matching N_2 binary descriptors of the target image is proportional to N_2. Our method thus has a time complexity of $O(n)$. If the number of key-points increases, our CAM-based matching becomes faster than the basic Hamming distance calculation method.

6.2.2 Speed Comparison

Table 6.1 compares our binary descriptor matching method with other published work in terms of speed. Since the contributions of [62]–[66] as shown in Table 6.1 have been primarily focused on the detector and descriptor stages of image matching, they have implemented the detector, binary descriptor, and binary descriptor matching steps of the image matching algorithm. Our focus and contributions are related to the matching step and to demonstrate a fair comparison, we compare with the works that use Hamming distance for the binary descriptor matching.

The focus of our work is on the binary descriptor matching step. Our proposed method can be used with any detector and binary descriptor algorithm and the matching stage is independent of the image size. The number of key-points and the number of bits directly affect performance and resource utilization. We compute and report the required time of applying our binary descriptor matching method to 128-bit (BRIEF), 256-bit (ORB), and 512-bit (BRISK) descriptors so that our results can be compared with other work that employ these three binary descriptors. As shown in Table 6.1, our method is faster than other work that uses the same number of key-points and descriptor bits. This is because our method does not require computation of the Hamming distance for all combinations of descriptors of the reference image and target image. Although the image size has a direct effect on the detector and descriptor speed and the frame rate metric, the processing speed of the matching step is only affected by the number of key-points and the binary descriptor size, and is independent of the image size. Therefore, since our focus and contributions in this work are on the matching step, and are not related to a specific detector and descriptor, we do not report the image size of our work in Table 6.1.

Rao et al. [62] report 3 ms as the latency of their binary descriptor matching step and 13.37 ms as the latency of the overall image matching algorithm. They compute the Hamming distance for 500 key-points per image. Their method requires 250,000 Hamming distance calculations for computing the distance between every two descriptors from reference and target images. In our design, 500 key-points lead to 500 operations and can be done in 500 clock cycles. This requires 0.005 ms at a frequency of 100 MHz which is 600 times faster in comparison with the binary descriptor matching step proposed in [62].

Table 6.1: Comparison of binary descriptor matching speed of our design with other work

Work	Descriptor	Number of bits	Number of key-points	Image size	Frequency (MHz)	Frame rate (fps)	Reported descriptor matching latency (ms)	Reported image matching latency (ms)
Huang et al. [63]	BRIEF	128	100	512×512	100	310	N/P***	3.2*
Ni et al. [64]	BRIEF	128	N/P***	640×480	100	162	N/P***	6.17*
This work	BRIEF	128	100	N/A**	100	N/A**	0.001	N/A**
Hu et al. [66]	BRIEF	128	500	640×1080	171	1306	N/P***	0.8083
Rao et al. [62]	ORB	256	500	1920×1080	150	N/P***	3	13.37
This work	ORB	256	500	N/A**	100	N/A**	0.005	N/A**
Peng et al. [65]	BRISK	512	4588	5616×3744	100	N/P***	548	N/P***
This work	BRISK	512	4588	N/A**	100	N/A**	0.045	N/A**

*The latency was not reported and is computed based on the reported frame rate.
** Not applicable (The matching stage is not dependent on image size and the latency is affected only by the number of bits and the number of key-points.)
***Not provided.

6.2.3 Accuracy of Image Matching

For verification of our hardware implementation, we obtain ORB descriptors from the images of the HPatches dataset [82], which is one of the most well-known and commonly-used datasets for descriptor matching in the literature. Our contribution is a method using a CAM architecture for descriptor matching which results in higher speed than seen in other work, while having similar *Precision* and *Recall*. Note that the purpose of accuracy verification in this section is to demonstrate that our method has similar accuracy performance with conventional matching methods used in the literature. Our CAM-based binary descriptor matching method has two parameters that can be tuned for a potential application. The first parameter is the summation threshold. After selecting the maximum of summation values (Sum_values), we compare the maximum value (Max_val) with a pre-defined threshold ($Sum_val_threshold$). The second parameter is the Hamming distance threshold ($Hamming_threshold$) for the final test of the selected pair. Fig. 6.7 and Fig. 6.8 illustrate *Precision* and *Recall* based on tuning these two parameters on the HPatches dataset. *Precision* and *Recall* evaluation metrics, which are common metrics for binary descriptor matching, are used in this work to validate the accuracy performance of our method. The *Recall* in this experiment is computed as the number of correct proposed matches divided by the number of matches in all known key-points for each pair of images according to (6.2).

$$Recall = \frac{\#\text{ of correct proposed matches}}{\#\text{ of matches of known key-points}} \tag{6.2}$$

The *Precision* value shows the ratio of correct matches from the proposed matches according to (6.3).

$$Precision = \frac{\#\text{ of correct proposed matches}}{\#\text{ of all proposed matches}} \tag{6.3}$$

In Fig. 6.7, we investigate the effect of the summation threshold before the Hamming distance unit (which is equivalent to having the $Hamming_threshold = 1$). As shown in Fig. 6.7, increasing the summation threshold ($Sum_val_threshold$), selects fewer key-points as potential matches in our design. As a result, *Precision* increases since only matches with higher similarities are accepted. On the other hand, *Recall* decreases if many of the correct matches are missed. The reason for the saturation of *Precision* and *Recall* at summation thresholds ($Sum_val_threshold$) lower than 25 is that we select the maximum value (Max_val), as shown in Algorithm 3. The $Sum_val_threshold$ is a parameter used for controlling the *Precision* and *Recall* of the matching results based on the requirements of an application. This value can be reasonably chosen based on the number of bits and the number of CAM modules. Although Figure 9 is provided for the purpose of explanation of the effect of changing this parameter on *Precision* and *Recall* values, the same methodology can be used on an application-specific dataset (on a training set for example).

Fig. 6.8 shows the effect of tuning the Hamming distance threshold ($Hamming_threshold$) on *Precision* and *Recall*. The thresholds can be reasonably chosen based on the number of descriptor bits, number of CAM units, and the requirements of a specific application. In this experiment, the summation threshold ($Sum_val_threshold$) is set to 25 empirically, based on Fig. 6.7. Lower Hamming distance threshold ($Hamming_threshold$) values lead to higher *Precision* as it results in matches with lower distance values. Increasing the Hamming distance threshold

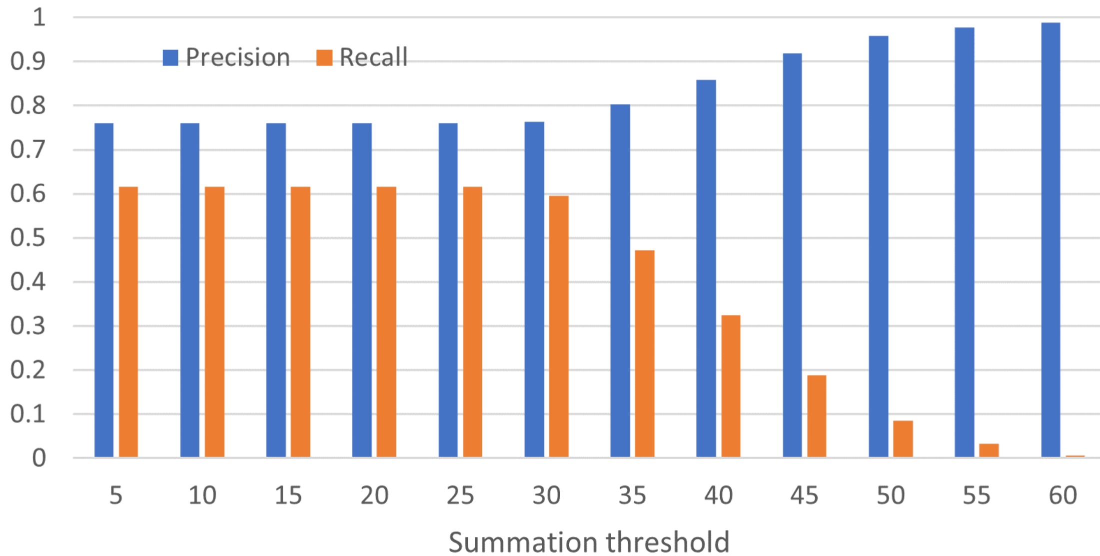

Figure 6.7: The effect of changing summation threshold on precision and recall of matching.

($Hamming_threshold$) results in a higher number of correct matches, which increases the *Recall*. Increasing the Hamming distance threshold ($Hamming_threshold$) more than 0.225 does not have any effect on *Precision* and *Recall* as all of the proposed matches pass the Hamming distance test. The value of 0.225 was set empirically based on Fig. 6.8 as the Hamming distance threshold ($Hamming_threshold$) to have higher *Precision* and *Recall*. If we do not add the final Hamming distance test to the end of our image matching pipeline, the *Precision* and *Recall* are equal to the highest values on the right end of Fig. 6.8.

The two parameters in Fig. 6.7 and 6.8 both have a linear effect on *Precision* and *Recall* values. Lower $Sum_val_threshold$ and higher $Hamming_threshold$ both lead to lower *Precision* and higher *Recall* while higher $Sum_val_threshold$ and lower $Hamming_threshold$ reduce the number of proposed matches and increase *Precision* and decrease *Recall*. Having these two control parameters is beneficial for reducing false positive matches while limiting the number of false negative matches which can be useful for various applications based on their requirements.

Descriptor matching is concerned with finding the nearest descriptor vector from a set of descriptors from a reference image to a query descriptor from the target image. Therefore, the conventional descriptor matching method which is generally used in the literature can be formulated as a K-nearest neighbor (KNN) problem. In the basic form, K is equal to 1 and the closest descriptor with a minimum distance (1-nearest neighbor) is selected. An improved version which is commonly used in the literature [88]–[90], uses $K = 2$ and selects the two nearest neighbor and performs a ratio test based on the calculated distances. As a result, we compare our proposed method to the conventional KNN method to demonstrate comparable accuracy of our novel CAM-based proposed matching method.

Table 6.2 shows the comparison between our CAM-based matching method with K-nearest neighbor (KNN) that uses Hamming distance as the distance metric. We used the KNN method (in this

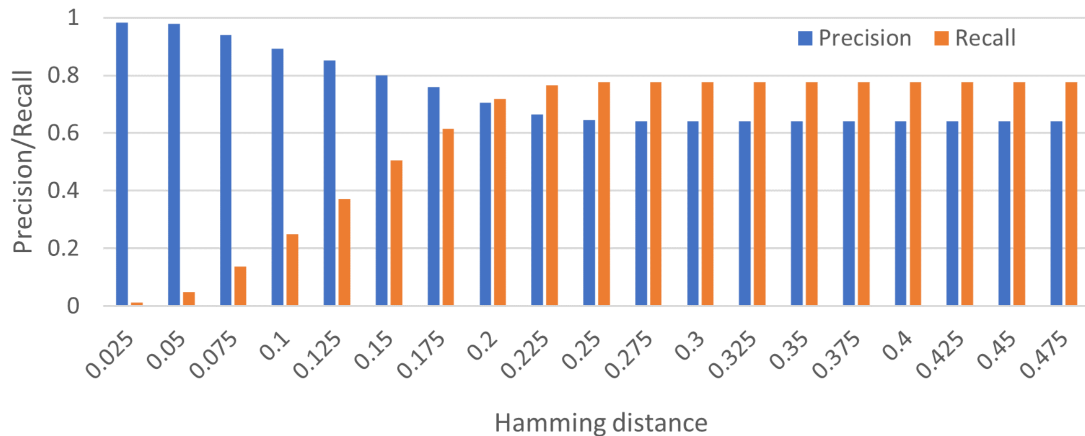

Figure 6.8: The effect of changing Hamming distance threshold on precision and recall of matching.

experiment K is equal to 2 so that the ratio test is applicable) to select the two best matches (lowest Hamming distances) for each key-point from the reference image and applied the ratio test proposed by Lowe [35] (one of the most well-known works in descriptor matching in the literature) to decide if the pair should be proposed as a match. The *Precision* and *Recall* shown in this table correspond to the Hamming distance threshold ($Hamming_threshold$) of 0.225 and the summation threshold ($Sum_val_threshold$) of 25. As shown in Table 6.2, our method exhibits *Precision* and *Recall* similar to that of the KNN method, indicating its performance is comparable to other commonly used matching methods.

Table 6.2: Comparison of precision and recall metrics

Matching method	**Precision**	**Recall**
KNN	0.7208	0.6209
Our method	0.7596	0.6157

The values of *Precision* and *Recall* are dependent on several factors such as the complexity of the dataset and the number of points in the experiment. But under identical conditions, as in Table 6.2, we show that our method achieves similar *Precision* and *Recall* for the same set of points with the KNN algorithm. It is important to note that any detector or descriptor algorithms can be used as the steps prior to descriptor matching in the image matching pipeline. But since the emphasis of our work is on matching, our experiments are focused on the matching step. Related work in the literature have used KNN, which is a commonly used descriptor matching technique, and as a result, have provided minimal detail on their matching step. Our method performs the same number of comparisons as the conventional method which leads to similar results (using the same descriptor vectors) as the conventional method in terms of *Precision* and *Recall*.

6.2.4 Hardware Implementation Metrics

All results provided in this section are based on the metrics after place and routing on the FPGA device. Table 6.3 illustrates the resource usage of our binary descriptor matching method. The reported results in Table 6.3 assume 100 key-points per image. In the LUT-CAM implementation, the CAM units are implemented using LUT RAM, and in BRAM-CAM implementation, the CAM units are implemented on BRAMs. We also report the resource usage of the binary descriptor matching step proposed by Ni et al. [64] in Table 6.3 as this is the only work, to the best of our knowledge, that reports resource utilization of the binary descriptor matching step. Since most of the previous work such as [63], [65], and [62] does not provide information about resource requirements of the matching stage, we cannot properly compare parameters such as power and resource utilization. Our method for 128-bit BRAM-based CAM requires about twice the number of LUTs and BRAMs as does [64]. However, because of our CAM-based binary descriptor matching, we achieve higher speed as a trade-off with resource utilization.

Table 6.3: Resource utilization for various number of bits

Configuration	LUT	LUT RAM	FF	BRAM
128-bit (LUT-CAM)	10274	1664	1487	2
256-bit (LUT-CAM)	20382	3328	1708	4
512-bit (LUT-CAM)	28441	5248	2573	7.5
128-bit (BRAM-CAM)	7926	64	1339	50
256-bit (BRAM-CAM)	15774	128	1668	100
512-bit (BRAM-CAM)	23006	256	2132	151.5
Ni et al. [64] (128-bit)	3479	26	2134	22.5

* Each BRAM unit contains 36 kb.
* Each LUT RAM unit contains 64 bits.

Table 6.4 presents the comparison of resource utilization for 100 and 500 number of key-points. As shown in Table 6.4, increasing the number of key-points (N in Fig. 6.3) leads to a higher number of bits for each CAM unit. Therefore, the number of adder units in Fig. 6.3 (a) and the number of comparator units, and the critical delay path of the comparator tree also increase.

Although the number of binary descriptor bits and the number of key-points extracted from each image effect both resource utilization and the *Precision* and *Recall* metrics of our proposed architecture, these numbers do not impact our contribution to speed improvement described in this work. Choosing a higher number of bits for the descriptors normally increases performance in terms of precision and selecting a higher number of key-points usually improves recall (depending on the data set). Each of these choices will increase the resource utilization as well. However, these design decisions are primarily related to the specific dataset and intended image matching application.

Table 6.5 shows the power and maximum operating frequency of various configurations of our CAM-based binary descriptor matching implementation. In Table 6.5, the numbers of bits (128, 256, 512) is proportional to the number of CAM units k (32, 64, 128) as the size of the CAM units is fixed (m =4). The maximum frequency is computed based on the critical timing path of the circuit. The maximum frequency for 100 key-points per image for varying number of bits does

Table 6.4: Resource utilization with various number of key-points

Configuration	Number of key-points	LUT	LUT RAM	FF	BRAM
LUT-CAM	100	20382 (8.4%)	3328 (2.9%)	1708 (0.4%)	4 (0.7%)
LUT-CAM	500	88492 (36.5%)	16212 (14.4%)	7667(1.6%)	4 (0.7%)
BRAM-CAM	100	15774 (6.5%)	128 (0.1%)	1668 (0.3%)	100 (16.7%)
BRAM-CAM	500	67464 (27.8%)	128 (0.1%)	7344 (1.5%)	452 (75.3%)

* Each BRAM unit contains 36 kb.
* Each LUT RAM unit contains 64 bits.

not lead to noticeable changes. This shows that the critical timing path of the design does not change significantly with changing the number of bits. However, using 500 key-points per image decreases the maximum operating frequency to 102.75 MHz and 91.69 MHz for BRAM-CAM and LUT-CAM, respectively. The change in the maximum operating frequency demonstrates that the critical timing path of the design is related to the comparator tree which is a combinational circuit with a size proportional to the number of key-points. In addition, the power consumption of the implementation with 500 key-points is much more than the implementation that uses 100 key-points, due to the usage of more BRAM as shown in Table 6.5.

The choice of using LUTs or BRAM for the implementation of CAM depends on the available resources, maximum operating frequency, and power consumption of the FPGA.

Table 6.5: Power usage and maximum operating frequency for various number of bits and key-points

Configuration	Number of key-points	Number of bits	Total power (W)	Maximum frequency (MHz)
LUT-CAM	100	128	0.967	124.44
LUT-CAM	100	256	1.386	125.79
LUT-CAM	100	512	1.719	123.82
BRAM-CAM	100	128	1.571	126.94
BRAM-CAM	100	256	1.398	126.44
BRAM-CAM	100	512	1.995	123.29
LUT-CAM	500	256	4.652	91.68
BRAM-CAM	500	256	10.018	102.75

Our proposed CAM matching method can be applied to any image size and used with any binary descriptor. As a case study, we provide the implementation metrics of a key-point description and matching, for two full-HD 1920×1080 images on the KCU105 FPGA board [87]. The resource utilization and speed metrics for 500 key-points per image and a 256-bit binary descriptor extracted from image patches of 65×65 are provided in Table 6.6.

The key-point locations are stored in RAM for each image. For each image, a 1920×65 line buffer is implemented so that the binary descriptor has parallel access to the pixel values. The descriptors from the first image are stored in CAM and the descriptors from the second image are

stored in a RAM. After computing all descriptors, the descriptors from the RAM (second image) are loaded sequentially, and the address of the best matching descriptor (first image) is retrieved from CAM. The address is used to load the matching key-point coordinates from the key-point memory. This implementation results in a maximum frequency of 77 MHz and a frame rate of 40 fps, which includes image loading and descriptor computation time as well as the time for matching.

Table 6.6: Resource utilization metrics for 500 key-points and 1920×1080 images on KCU105

Resources	Utilization	Available	Utilization (%)
LUT	97083	242400	40
LUTRAM	16552	112800	14.67
FF	78432	484800	16.17
BRAM	74	600	12.33

6.3 Conclusion

In this chapter, we introduce a method for using CAM to increase the speed of binary descriptor matching. Our method finds the location of the closest descriptor stored in a data memory to a query descriptor and proposes these descriptors' corresponding key-points as a match. We also present an FPGA-based hardware architecture for binary descriptor matching based on partitioned CAM. The architecture can be used with any local binary descriptor to accelerate the binary descriptor matching step while maintaining similar *Precision* and *Recall* to that of KNN. We anticipate *Precision* and *Recall* can be improved with optimization. Our method is faster than other conventional binary descriptor matching methods while consuming more power and having higher resource utilization due to the usage of CAM. We suggest investigating using CAM for non-binary descriptor matching such as SIFT and SURF as well as improving *Precision* and *Recall* as future path of this work.

Chapter 7

Real-time Aerial Image Matching using Circular Shifting Binary Descriptors and CAM-based Matching

In this chapter, we demonstrate an application case study of our proposed approaches described in previous chapters. We use a circular shifting binary descriptor and CAM-based matching for finding the matching pairs. We compute the matches on aerial images captured over the University of Victoria using an unmanned aerial vehicle (UAV) and compare the accuracy and speed with a conventional matching method to evaluate our proposed approach. In Section 7.1, we present our experimental setup and the data set used in this chapter. Then, we provide the results of our experiments in section 7.2. Finally, we conclude this chapter in section 7.3.

7.1 Experimental Setup

In this section, we present the experimental setup for applying our image matching method on aerial images. The data set used in this section is a set of images captured over University of Victoria campus in 2021 by an unmanned aerial vehicle (UAV). The image data is captured by movements of the UAV in yaw, roll, and pitch rotations (as shown as Fig. 7.1) and zooming in the forward direction. We use two image sets which consist of the Bob Wright Building and the Continuing Studies Building in these experiments. Figure 7.2 shows examples of images and various movements in this data set.

We separate the sequence of images captured from the UAV into groups each having $k = 6$ images. The number of correct matches are computed and the mean Average Precision metric is computed for each group of the sequence.

Figure 7.3 presents the overall block diagram of our proposed approach for aerial image matching.

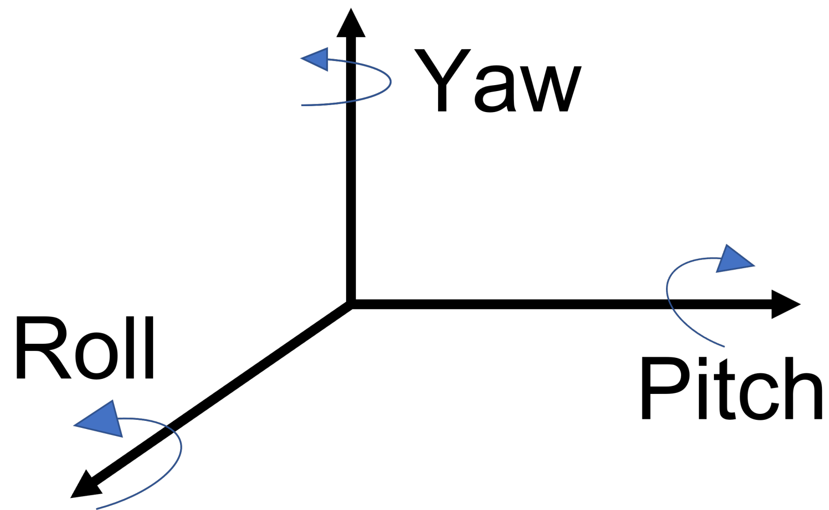

Figure 7.1: Pitch, yaw and roll angles.

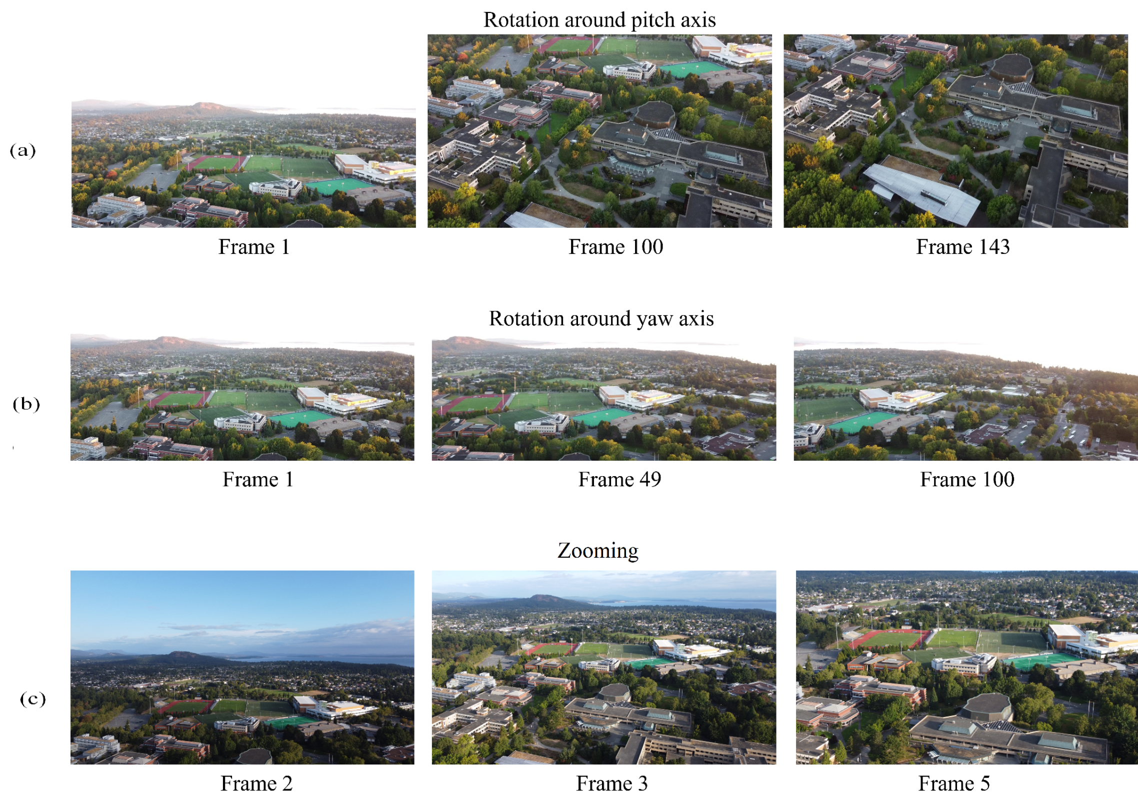

Figure 7.2: Example image frames of various UAV rotations and movements of the Continuing Studies Building at the University of Victoria.

In this approach, we use a circular shifting binary descriptor in combination with CAM to find the matched pairs of points between images. As shown in Fig. 7.3, first, the key-points are detected in

the two images. For detecting the key-points, we use the FAST algorithm. The FAST algorithm compares the intensity of the pixel in the center of the image with pixels on a circle with a fixed predefined radius around that pixel. Then, the descriptor vectors are computed for each of the detected key-points. We use the circular shifting binary descriptor which was explained in Chapter 5 for generating the descriptor vectors. Next, we store the descriptors generated from the first image in a CAM. To achieve the rotation invariance property for the descriptor vectors, we compare every circular shifted version of the descriptors computed from the second image with the descriptor from the first image. The rotated descriptor block in Fig. 7.3 generates the shifted descriptors by changing the placement of bits of the computed descriptors. Our design supports 5 degrees of rotation which supports 72 rotations for a 256-bit descriptor. In the last step, the results of matching from the CAM architecture (as described in Chapter 6) is generated as the output of our proposed system.

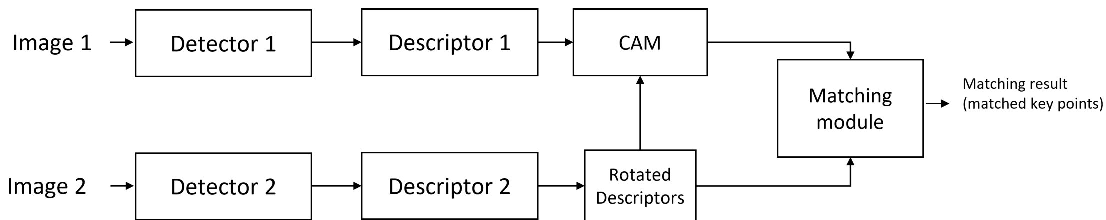

Figure 7.3: Overall block diagram of our proposed approach for combination of a circular shifting binary descriptor and CAM matching.

Figure 7.4 shows our experimental setup for the evaluation of aerial images. We compare our proposed method with the ORB algorithm as a baseline which is a commonly-used and well-known binary descriptor algorithm. For this experiment, we select one frame (known as the key frame) as a reference image and compute the matches from that image to $k-1$ consecutive images. After that, the next key frame, which is frame number $k+1$, is selected as the reference image of the next set and is matched with the next $k-1$ images following that. For each group of k images, the mean Average Precision value is computed. In our experiment, we chose $k=6$ for both ORB algorithm and our approach.

For generating the ground truth matchings, first, we use the SIFT algorithm [35] to propose an initial set of matches. The homography matrix is used to decide if the matches generated by the ORB algorithm and our method are correct. Finally, we can compute the mean Average Precision metric for each set of k consecutive frames for both algorithms. In the next section, we present the results of our experiment.

7.2 Experimental Results

In this section, we provide the results of experiment for evaluation of our proposed method on the aerial image dataset. Figure 7.5 shows the comparison of mAP of our proposed method and the ORB algorithm for various consecutive key-frames. The results demonstrate that our method attains results similar to that of the ORB algorithm.

Our method requires $I_W \times I_H$ clock cycles (where I_W and I_H are the width and height of the image, respectively) to process the whole image and an additional $N_{\text{key-points}} \times N_{rotations}$ clock cycles

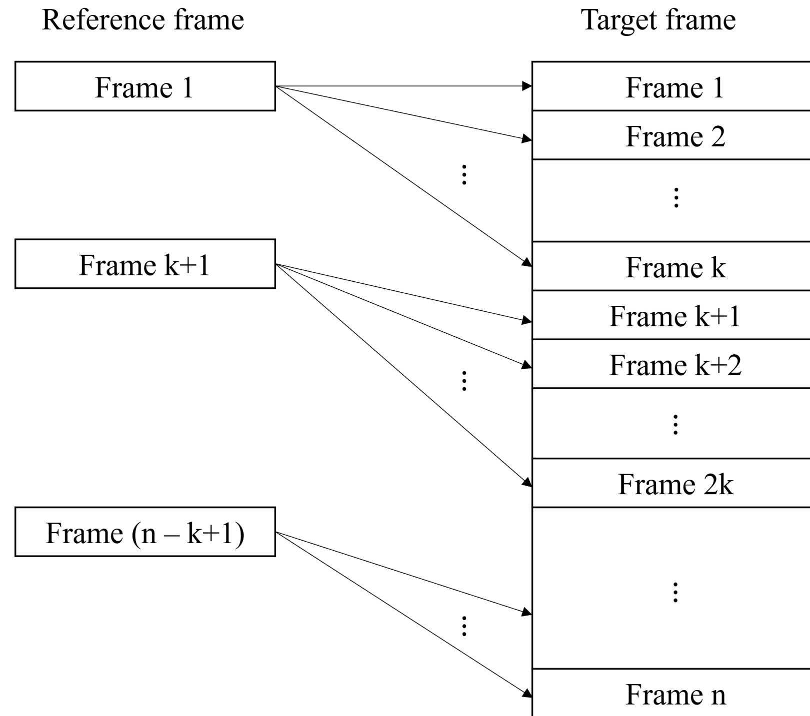

Figure 7.4: Key frame selection and image grouping for evaluation of image matching on consecutive images.

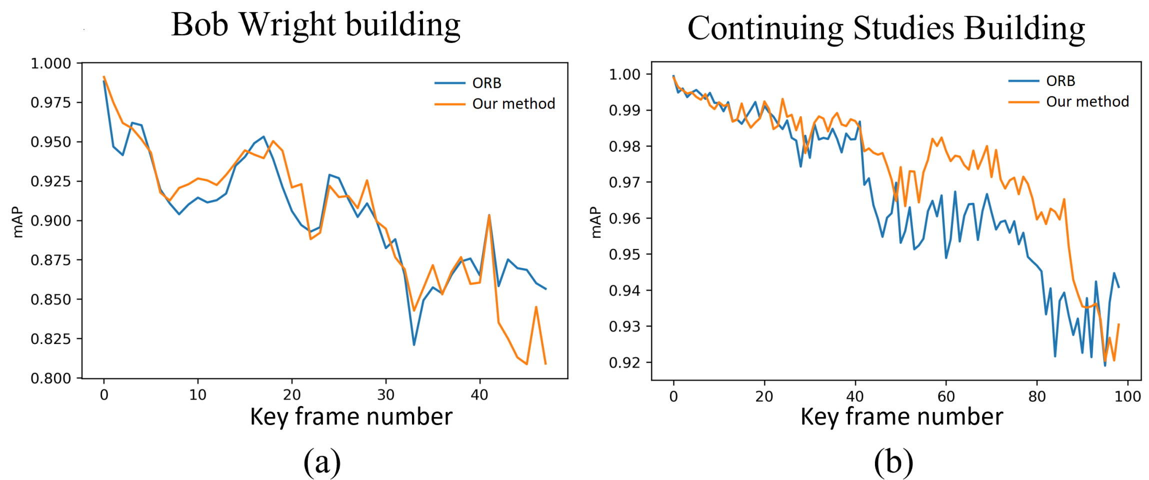

Figure 7.5: Comparison of mAP of our proposed method with the ORB algorithm.

are required to find all the matches using CAM-based matching. Therefore, since $N_{rotations} = 72$ in our method, the overall required number of clock cycles for processing the matching of two frames is as shown in (7.1):

$$C_{\text{proposed method}} = I_H \times I_W + N_{\text{key-points}} \times 72 \tag{7.1}$$

An estimate of the overall clock cycles required for the implementation of ORB description and Hamming distance matching is shown in (7.2). In this estimate, $I_W \times I_H$ clock cycles are required to process the image similar to our descriptor method, but for the key-point matching step, each descriptor from the first image should be compared with each descriptor from the second image.

$$C_{ORB} = I_H \times I_W + N_{\text{key-points}} \times N_{\text{key-points}} \tag{7.2}$$

As an example, if $N_{\text{key-points}} = 500$, $I_W = 1920$, and $I_H = 1080$, 2,109,600 clock cycles are required to process two frames using CAM matching, which leads to 47 frames per second at 100 MHz. For the conventional ORB matching method, about 2,323,600 clock cycles are required which results in 43 frames per second.

To illustrate the speed enhancement of our proposed method, we calculate the frame rate of our design and the frame rate of the ORB algorithm with conventional Hamming matching for various numbers of key-points. Figure 7.6 shows the comparison of frame rate for various number of key-points. The frame rate decreases faster for an ORB+Hamming matching approach in comparison with our approach (circular shifting binary descriptor+CAM matching).

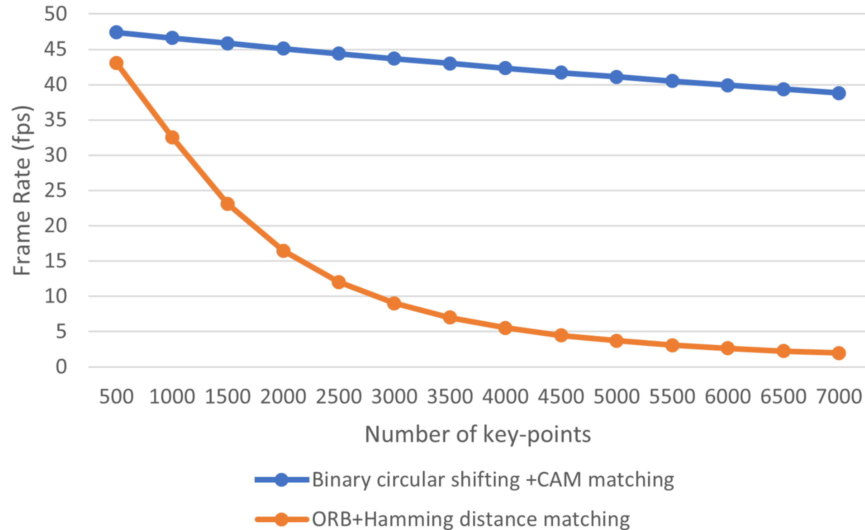

Figure 7.6: Comparison of frame rate vs. number of key-points.

Figure 7.7 shows example images and the matches proposed by our method using the Continuing Studies Building images. The movement of the camera is from top to bottom (pitch axis). In each row of Fig. 7.7, the left image is a key frame and the right image is the first image after the key frame. The green lines represent the correct matches while the red lines represent incorrect matches proposed by our method based on the ground truth.

7.3 Conclusion

In this chapter, we applied our proposed methods for the description and matching steps to an aerial image dataset and evaluated the performance of our proposed methods in a real world application. Our experimental results demonstrate that our method has similar accuracy with respect to the ORB algorithm and Hamming distance matching, and achieves faster matching results as the number of key-points increases.

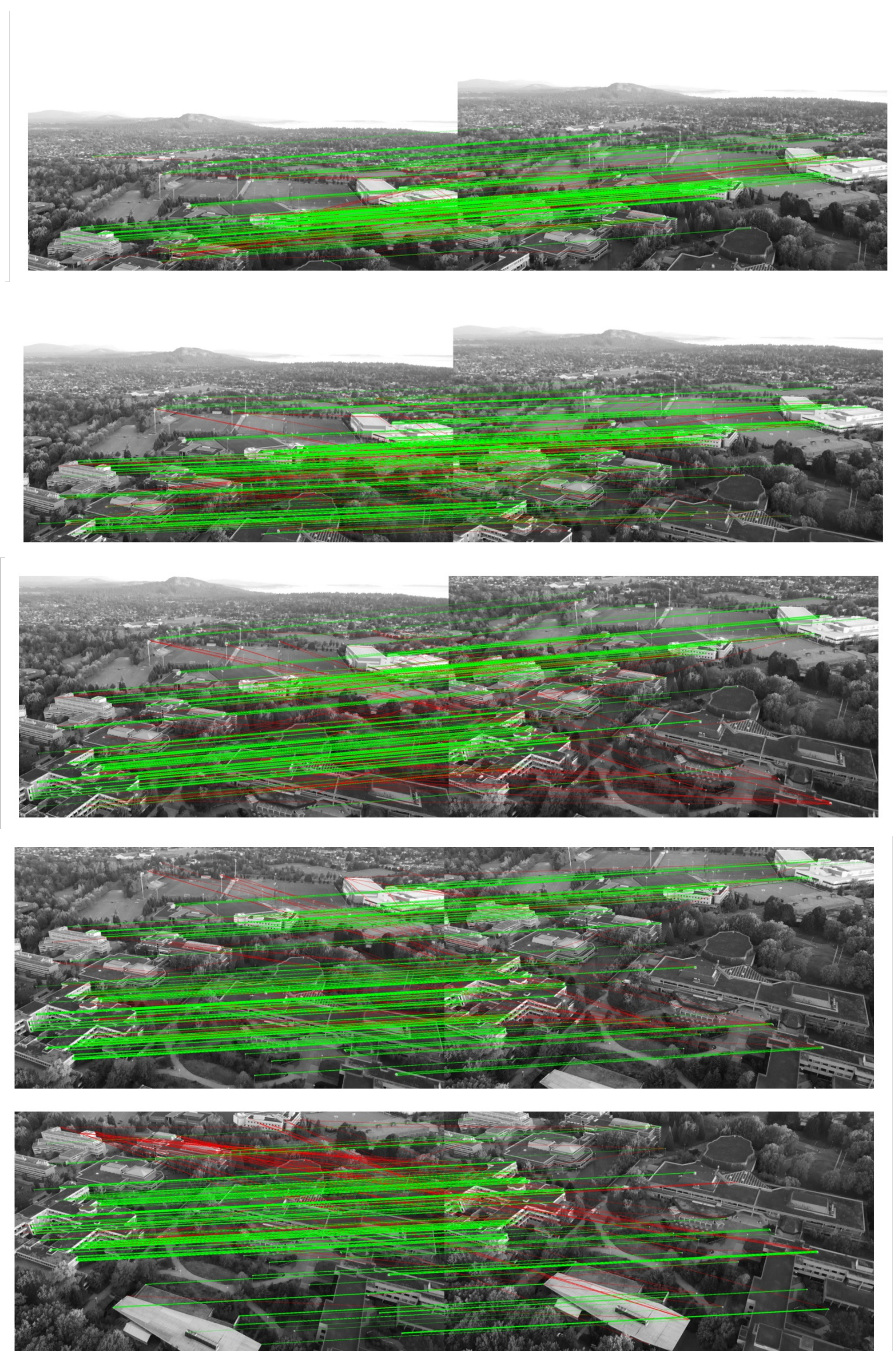

Figure 7.7: Example matching of consecutive frames of the Continuing Studies Building images. The green lines show the correct matches. The red lines are incorrect matches. These images are best viewed in color and zoomed in.

Chapter 8

Conclusion

The main contributions of this dissertation are proposing algorithmic and hardware design methods for improving the speed of computer vision algorithms. In Chapter 2, related work in the literature were reviewed. Chapter 3 focused on the hardware acceleration of the HOG algorithm. Chapter 4 presented an FPGA-based implementation of the AKAZE algorithm which accelerates the non-linear scale-space generation step in image matching algorithms. In Chapter 5, a new binary descriptor based on circular shifting of the descriptor vector was introduced which is shown to lead to efficient rotation invariant image matching. A modified CAM architecture was presented in Chapter 6 for improving the speed of the matching step of the image matching pipeline. Finally, an application of the proposed contributions of this dissertation on aerial image matching was presented in Chapter 7.

8.1 Summary of Contributions

In Chapter 3, a hardware–software co-design system of the HOG algorithm was introduced which extracts the HOG features and makes a decision based on those features. Our research resulted in four main contributions on HOG implementation. The first contribution, which was at the task allocation level, defines a well-organized partitioning between the various parts of a hardware–software co-design system. This partitioning results in the consumption of fewer FPGA resources than comparable hardware–software solutions. The second contribution in this work, which proposes a logarithm-based bin assignment in the HOG algorithm. This bin assignment technique results in a multiplier-free implementation of the HOG algorithm and reduces the overall number of multipliers for the HOG-SVM core. Reducing the number of FPGA resources such as multipliers is advantageous due to the limited number of resources on chip, especially in more complex and larger designs. The third contribution is demonstrating how two parallel histogram computation modules and switching between them saves one clock cycle out of each 9 cycles of processing the image. As a result, the HOG core can accommodate the pixel data in a streaming manner on each clock cycle without any pause. Finally, a simpler implementation of the block normalization step was proposed in Chapter 3 for reducing the utilization of the FPGA resources.

The remainder of the contributions in this dissertation are related to the different steps of the

image matching algorithms. For the scale-space generation step, a design for non-linear scale-space generation for the AKAZE algorithm was proposed. Using a non-linear scale-space the AKAZE algorithm leads to higher accuracy but requires more computations. The contributions presented in Chapter 4 are summarized as follows.

First, based on the steps of the AKAZE algorithm which uses two passes through the image, the idea of using four parallel channels to generate a non-linear scale-space was introduced. Considering the fact that in the first step, the image is loaded once from the external memory to on-chip memory, different sections of the image can be accessed simultaneously. A memory management unit was introduced to store the image in four separate block RAMs so that sub-levels of each section of the image are generated in parallel. This contribution leads to a noticeable speed up in AKAZE non-linear scale-space implementation.

Second, an architecture was proposed for the second octave line buffers. The resources in this architecture are based on the same data path as the first octave, but in a different scale. A multi-scale line buffers with several output windows for parallelizing the image input at different scales was introduced. By using the proposed line buffers, the architecture consumes about half of the line buffer registers in comparison with other architecture reported in the literature.

The third contribution of the AKAZE implementation is the design of a time-sharing mechanism in the memory management unit to process different sections of the image in parallel with access to the required pixels from the other sections. By using the proposed time-sharing mechanism, multiple sections of the image (which are stored in separate memory banks) are processed in parallel. The overall results of these contributions result in a frame rate of 304 frames per second for 1280×768 image resolution which is the highest frame rate in the same image resolution and same frequency in comparison with other work.

Further contributions in this research are related to the descriptor stage for image matching algorithms. In Chapter 5, a method was proposed for enhancing the speed of calculation of rotation invariant binary descriptors. In this method, a sample seed is used for generating a sampling pattern by rotating the sample seed around each key-point. This allows skipping computation of the orientation in the description step. Instead, the generated descriptor is circularly rotated in the matching step so that the best rotation of the image patch is selected for matching. It was demonstrated that the number of operations in this proposed method is less than other methods, rotation error of the proposed method is similar to other published work while maintaining comparable image matching accuracy.

In Chapter 6, a new method for using CAM to increase the speed of binary descriptor matching was introduced. As described in Chapter 6, the location of the closest descriptor stored in a data memory to a query descriptor is found using a new CAM architecture and corresponding key-points are introduced as a match. An FPGA-based hardware architecture for binary descriptor matching based on partitioned CAM is also presented in Chapter 6. The proposed architecture can be used with any local binary descriptor to accelerate the binary descriptor matching step while maintaining similar precision and recall. The proposed method is faster than other conventional binary descriptor matching methods although it requires more power and resource utilization due to the usage of CAM.

Finally, in Chapter 7, a practical application to aerial image processing using the proposed methods in this dissertation was presented. Aerial image processing typically requires a high number

of computations due to the large number of key-points in each image. In Chapter 7, a comparison of mean Average Precision between circular shifting binary descriptor with CAM matching and the ORB descriptor with Hamming matching was presented. It was demonstrated that the proposed methods in this dissertation accelerate the image matching computation without accuracy loss.

8.2 Future Work

In this section, possible additional research avenues of the work presented in this dissertation are suggested.

Chapter 3 introduced a new hardware-software co-design of the HOG algorithm. This could be implemented as several parallel IP-cores to process a single image. One of the future paths of this research would be to modify the design to take advantage of this feature and enhance processing speed. Another possibility is to design efficient interrupt mechanisms efficiently for reading precomputed window addresses from the memory. In this way, the processor can perform other tasks while the HOG-SVM cores and DMAs are processing the image. Additional opportunities to continue this research includes the use of the proposed methods to implement variants of the HOG algorithm. Other variants of the HOG algorithm, such as HOG-3d [78], require a large number of computations and can benefit from parallel implementation.

In Chapter 4, an implementation of non-linear scale-space generation of the AKAZE algorithm was presented. The AKAZE algorithm is based on fast explicit diffusion. There are other diffusion algorithms that can be investigated as a possible future avenue for this research. In addition, their suitability for hardware implementation can be assessed. Another possible next step for this research is to consider different detectors and descriptors that can be added to the current architecture, following the parallel channel processing concept introduced in Chapter 4.

A potential future avenue of research for on-going research on circular shifting binary descriptors, which was presented in Chapter 5, is focusing on increasing the matching accuracy by tuning the location of sampling patterns. For this purpose, search optimization algorithms can be used to find an optimal pattern which results in maximum matching accuracy. Although the circular shifting binary descriptor was shown to have less number of operations than other binary description algorithms, it can also be implemented in parallel for attaining even higher speeds using hardware platforms such as FPGAs.

In Chapter 6, a new modified CAM architecture was introduced for binary descriptor matching. Binary descriptors have advantages that include less memory footprint requirements and higher computation speed in comparison with non-binary descriptors. However, non-binary descriptor algorithms can achieve higher accuracy results than binary descriptor algorithms. Using CAM for non-binary descriptors matching (such as SIFT and SURF) is a potential future paths of this work. If the proposed CAM architecture were to be be modified to accommodate non-binary descriptors, the speed of non-binary descriptor matching will increase.

References

[1] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, 886–893 vol. 1. DOI: 10.1109/CVPR.2005.177.

[2] M.-E. Ilas, "New histogram computation adapted for FPGA implementation of HOG algorithm: For car detection applications," in *2017 9th Computer Science and Electronic Engineering (CEEC)*, 2017, pp. 77–82. DOI: 10.1109/CEEC.2017.8101603.

[3] J. Li, X. Liu, F. Liu, D. Xu, Q. Gu, and I. Ishii, "A Hardware-Oriented Algorithm for Ultra-High-Speed Object Detection," *IEEE Sensors Journal*, vol. 19, no. 10, pp. 3818–3831, 2019. DOI: 10.1109/JSEN.2019.2895294.

[4] P. Loncomilla, J. Ruiz-del-Solar, and L. Martínez, "Object recognition using local invariant features for robotic applications: A survey," *Pattern Recognition*, vol. 60, pp. 499–514, 2016, ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2016.05.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320316301054.

[5] U. Sharif, Z. Mehmood, T. Mahmood, M. A. Javid, A. Rehman, and T. Saba, "Scene analysis and search using local features and support vector machine for effective content-based image retrieval," *Artificial Intelligence Review*, vol. 52, no. 2, pp. 901–925, Aug. 2019, ISSN: 1573-7462. DOI: 10.1007/s10462-018-9636-0. [Online]. Available: https://doi.org/10.1007/s10462-018-9636-0.

[6] W. Lyu, Z. Zhou, L. Chen, and Y. Zhou, "A survey on image and video stitching," *Virtual Reality & Intelligent Hardware*, vol. 1, no. 1, pp. 55–83, 2019, ISSN: 2096-5796. DOI: https://doi.org/10.3724/SP.J.2096-5796.2018.0008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2096579619300063.

[7] T. Taketomi, H. Uchiyama, and S. Ikeda, "Visual SLAM algorithms: a survey from 2010 to 2016," *IPSJ Transactions on Computer Vision and Applications*, vol. 9, no. 1, p. 16, Jun. 2017, ISSN: 1882-6695. DOI: 10.1186/s41074-017-0027-2. [Online]. Available: https://doi.org/10.1186/s41074-017-0027-2.

[8] G. Fahim, K. Amin, and S. Zarif, "Single-View 3D reconstruction: A Survey of deep learning methods," *Computers & Graphics*, vol. 94, pp. 164–190, 2021, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2020.12.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849320301849.

[9] J. Ma, X. Jiang, A. Fan, J. Jiang, and J. Yan, "Image Matching from Handcrafted to Deep Features: A Survey," *International Journal of Computer Vision*, vol. 129, no. 1, pp. 23–79, Jan. 2021, ISSN: 1573-1405. DOI: 10.1007/s11263-020-01359-2. [Online]. Available: https://doi.org/10.1007/s11263-020-01359-2.

[10] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "Architectural Study of HOG Feature Extraction Processor for Real-Time Object Detection," in *2012 IEEE Workshop on Signal Processing Systems*, 2012, pp. 197–202. DOI: 10.1109/SiPS.2012.57.

[11] M. Komorkiewicz, M. Kluczewski, and M. Gorgon, "Floating point HOG implementation for real-time multiple object detection," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2012, pp. 711–714, ISBN: 9781467322577.

[12] Z. Xiang, H. Tan, and W. Ye, "The Excellent Properties of a Dense Grid-Based HOG Feature on Face Recognition Compared to Gabor and LBP," *IEEE Access*, vol. 6, pp. 29 306–29 319, 2018. DOI: 10.1109/ACCESS.2018.2813395.

[13] M. Awais, M. J. Iqbal, I. Ahmad, *et al.*, "Real-Time Surveillance Through Face Recognition Using HOG and Feedforward Neural Networks," *IEEE Access*, vol. 7, pp. 121 236–121 244, 2019. DOI: 10.1109/ACCESS.2019.2937810.

[14] W. Xing, N. Deng, B. Xin, Y. Liu, Y. Chen, and Z. Zhang, "Identification of Extremely Similar Animal Fibers Based on Matched Filter and HOG-SVM," *IEEE Access*, vol. 7, pp. 98 603–98 617, 2019. DOI: 10.1109/ACCESS.2019.2923225.

[15] N. Laopracha, K. Sunat, and S. Chiewchanwattana, "A Novel Feature Selection in Vehicle Detection Through the Selection of Dominant Patterns of Histograms of Oriented Gradients (DPHOG)," *IEEE Access*, vol. 7, pp. 20 894–20 919, 2019. DOI: 10.1109/ACCESS.2019.2893320.

[16] M. Ehatisham-Ul-Haq, A. Javed, M. A. Azam, *et al.*, "Robust Human Activity Recognition Using Multimodal Feature-Level Fusion," *IEEE Access*, vol. 7, pp. 60 736–60 751, 2019. DOI: 10.1109/ACCESS.2019.2913393.

[17] K. Negi, K. Dohi, Y. Shibata, and K. Oguri, "Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm," in *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–8. DOI: 10.1109/FPT.2011.6132679.

[18] S. Ojha and S. Sakhare, "Image processing techniques for object tracking in video surveillance-A survey," in *2015 International Conference on Pervasive Computing (ICPC)*, 2015, pp. 1–6. DOI: 10.1109/PERVASIVE.2015.7087180.

[19] S. Ghaffari, P. Soleimani, K. F. Li, and D. W. Capson, "Analysis and Comparison of FPGA-Based Histogram of Oriented Gradients Implementations," *IEEE Access*, vol. 8, pp. 79 920–79 934, 2020. DOI: 10.1109/ACCESS.2020.2989267.

[20] S. Ghaffari, P. Soleimani, K. F. Li, and D. W. Capson, "A Novel Hardware–Software Co-Design and Implementation of the HOG Algorithm," *Sensors*, vol. 20, no. 19, 2020, ISSN: 1424-8220. DOI: 10.3390/s20195655. [Online]. Available: https://www.mdpi.com/1424-8220/20/19/5655.

[21] S. Ghaffari, P. Soleimani, K. F. Li, and D. Capson, "FPGA-based Implementation of HOG Algorithm: Techniques and Challenges," in *2019 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2019, pp. 1–7. DOI: 10.1109/PACRIM47961.2019.8985056.

[22] P. Soleimani, D. W. Capson, and K. F. Li, "Real-time FPGA-based implementation of the AKAZE algorithm with nonlinear scale space generation using image partitioning," *Journal of Real-Time Image Processing*, 2021. DOI: 10.1007/s11554-021-01089-9.

[23] P. Soleimani, K. F. Li, and D. W. Capson, "A circular shifting binary descriptor for efficient rotation invariant image matching," in *2022 26th International Conference on Pattern Recognition (ICPR)*, 2022, pp. 393–399. DOI: 10.1109/ICPR56361.2022.9956083.

[24] J. Luo and C. Lin, "Pure FPGA Implementation of an HOG Based Real-Time Pedestrian Detection System," *Sensors*, vol. 18, no. 4, p. 1174, 2018. DOI: 10.3390/s18041174.

[25] M. Qasaimeh, J. Zambreno, and P. H. Jones, "A Runtime Configurable Hardware Architecture for Computing Histogram-Based Feature Descriptors," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 351–3513. DOI: 10.1109/FPL.2018.00066.

[26] X. Long, S. Hu, Y. Hu, Q. Gu, and I. Ishii, "An FPGA-Based Ultra-High-Speed Object Detection Algorithm with Multi-Frame Information Fusion," *Sensors*, vol. 19, no. 17, p. 3707, 2019. DOI: 10.3390/s19173707.

[27] V. Ngo, D. Castells-Rufas, A. Casadevall, M. Codina, and J. Carrabina, "Low-Power Pedestrian Detection System on FPGA," *Proceedings*, vol. 31, no. 1, pp. 35–, 2019, ISSN: 2504-3900.

[28] X. Ma, W. A. Najjar, and A. K. Roy-Chowdhury, "Evaluation and Acceleration of High-Throughput Fixed-Point Object Detection on FPGAs," *IEEE transactions on circuits and systems for video technology*, vol. 25, no. 6, pp. 1051–1062, 2015, ISSN: 1051-8215.

[29] J. Rettkowski, A. Boutros, and D. Göhringer, "HW/SW Co-Design of the HOG algorithm on a Xilinx Zynq SoC," *Journal of Parallel and Distributed Computing*, vol. 109, pp. 50–62, 2017, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.05.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301569.

[30] S.-S. Huang, S.-Y. Lin, and P.-Y. Hsiao, "An FPGA-Based HOG Accelerator with HW/SW Co-Design for Human Detection and Its Application to Crowd Density Estimation," *Journal of Software Engineering and Applications*, vol. 12, no. 01, pp. 1–19, 2019. DOI: 10.4236/jsea.2019.121001.

[31] V. Ngo, A. Casadevall, M. Codina, D. Castells-Rufas, and J. Carrabina, "A High-Performance HOG Extractor on FPGA," *arXiv:1802.02187v1*, 2018. [Online]. Available: https://arxiv.org/abs/1802.02187.

[32] M. Bilal, A. Khan, M. U. Karim Khan, and C.-M. Kyung, "A Low-Complexity Pedestrian Detection Framework for Smart Video Surveillance Systems," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 10, pp. 2260–2273, 2017. DOI: 10.1109/TCSVT.2016.2581660.

[33] Z. Yu, S. Yang, I. Sillitoe, and K. Buckley, "Towards a scalable hardware/software co-design platform for real-time pedestrian tracking based on a ZYNQ-7000 device," in *2017 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 2017, pp. 127–132. DOI: 10.1109/ICCE-ASIA.2017.8307853.

[34] K. Mikolajczyk, T. Tuytelaars, C. Schmid, *et al.*, "A Comparison of Affine Region Detectors," *International Journal of Computer Vision*, vol. 65, no. 1, pp. 43–72, Nov. 2005, ISSN: 1573-1405. DOI: 10.1007/s11263-005-3848-x. [Online]. Available: https://doi.org/10.1007/s11263-005-3848-x.

[35] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004, ISSN: 1573-1405. DOI: 10.1023/B:VISI.0000029664.99615.94. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[36] P. F. Alcantarilla, A. Bartoli, and A. J. Davison, "KAZE Features," in *Computer Vision – ECCV 2012*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 214–227, ISBN: 978-3-642-33783-3.

[37] P. Alcantarilla, J. Nuevo, and A. Bartoli, "Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces," *Procedings of the British Machine Vision Conference 2013*, 2013. DOI: 10.5244/c.27.13.

[38] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded Up Robust Features," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417, ISBN: 978-3-540-33833-8.

[39] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking," in *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, vol. 2, 2005, 1508–1515 Vol. 2. DOI: 10.1109/ICCV.2005.104.

[40] C. Leng, H. Zhang, B. Li, G. Cai, Z. Pei, and L. He, "Local Feature Descriptor for Image Matching: A Survey," *IEEE Access*, vol. 7, pp. 6424–6434, 2019. DOI: 10.1109/ACCESS.2018.2888856.

[41] M. Calonder, V. Lepetit, M. Ozuysal, T. Trzcinski, C. Strecha, and P. Fua, "BRIEF: Computing a Local Binary Descriptor Very Fast," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 7, pp. 1281–1298, 2012. DOI: 10.1109/TPAMI.2011.222.

[42] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[43] X. Yang and K.-T. Cheng, "LDB: An ultra-fast feature for scalable Augmented Reality on mobile devices," in *2012 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2012, pp. 49–57. DOI: 10.1109/ISMAR.2012.6402537.

[44] C. Li, C. Zhu, J. Zhang, B. Luo, X. Wu, and J. Tang, "Learning Local-Global Multi-Graph Descriptors for RGB-T Object Tracking," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 29, no. 10, pp. 2913–2926, 2019. DOI: 10.1109/TCSVT.2018.2874312.

[45] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust invariant scalable keypoints," in *2011 International Conference on Computer Vision*, 2011, pp. 2548–2555. DOI: 10.1109/ICCV.2011.6126542.

[46] J. Zhong, Y. Li, L. Gu, Q. Wang, L. Li, and Y. Hu, "Robust local binary descriptor in rotation change using polar location," *Journal of Electronic Imaging*, vol. 30, no. 03, 2021. DOI: 10.1117/1.jei.30.3.031203.

[47] J. Zhong, Y. Li, Q. Wang, L. Zhang, and Y. Hu, "An Improved Local Binary Descriptor Based Polar Gridding for Rotation Change," in *Urban Intelligence and Applications*, ser. Communications in Computer and Information Science, Singapore: Springer Singapore, 2020, pp. 39–46, ISBN: 9789813346000.

[48] D. Zhang, H. Chen, F. Yin, Z. Chen, H. Tang, and H. Xu, "Efficient and distinctive binary descriptor for rotated circular image recognition," *Machine vision and applications*, vol. 30, no. 4, pp. 749–761, 2019, ISSN: 0932-8092.

[49] T. Ojala, M. Pietikainen, and D. Harwood, "Performance evaluation of texture measures with classification based on Kullback discrimination of distributions," in *Proceedings of 12th International Conference on Pattern Recognition*, vol. 1, IEEE, 1994, 582–585 vol.1, ISBN: 0818662654.

[50] F. Bellavia, D. Tegolo, and E. Trucco, "Improving SIFT-based Descriptors Stability to Rotations," in *2010 20th International Conference on Pattern Recognition*, 2010, pp. 3460–3463. DOI: 10.1109/ICPR.2010.845.

[51] F. Bellavia and C. Colombo, "Rethinking the sGLOH Descriptor," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 931–944, 2018. DOI: 10.1109/TPAMI.2017.2697849.

[52] V. Garcia, É. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," in *2010 IEEE International Conference on Image Processing*, 2010, pp. 3757–3760. DOI: 10.1109/ICIP.2010.5654017.

[53] B. Ramkumar, R. Laber, H. Bojinov, and R. S. Hegde, "GPU acceleration of the KAZE image feature extraction algorithm," *Journal of real-time image processing*, vol. 17, no. 5, pp. 1169–1182, 2019, ISSN: 1861-8200.

[54] G. Jiang, L. Liu, W. Zhu, S. Yin, and S. Wei, "A 127 fps in full HD accelerator based on optimized AKAZE with efficiency and effectiveness for image feature extraction," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6. DOI: 10.1145/2744769.2744772.

[55] L. Kalms, A. Elhossini, and B. Juurlink, "FPGA based hardware accelerator for KAZE feature extraction algorithm," in *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 281–284. DOI: 10.1109/FPT.2016.7929553.

[56] L. Kalms, K. Mohamed, and D. Göhringer, "Accelerated Embedded AKAZE Feature Detection Algorithm on FPGA," *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2017. DOI: 10.1145/3120895.3120898.

[57] N. Mentzer, J. Mahr, G. Payá-Vayá, and H. Blume, "Online stereo camera calibration for automotive vision based on HW-accelerated A-KAZE-feature extraction," *Journal of Systems Architecture*, vol. 97, pp. 335–348, 2019, ISSN: 1383-7621. DOI: `https://doi.org/10.1016/j.sysarc.2018.11.003`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S1383762118304740`.

[58] S. Ghaffari, D. W. Capson, and K. F. Li, "A Fully Pipelined FPGA Architecture for Multiscale BRISK Descriptors With a Novel Hardware-Aware Sampling Pattern," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 6, pp. 826–839, 2022. DOI: `10.1109/TVLSI.2022.3151896`.

[59] R. Sun, J. Qian, R. H. Jose, *et al.*, "A Flexible and Efficient Real-Time ORB-Based Full-HD Image Feature Extraction Accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 565–575, 2020. DOI: `10.1109/TVLSI.2019.2945982`.

[60] D.-H. Le, T. B. T. Cao, K. Inoue, and C.-K. Pham, "A fast CAM-based image matching system on FPGA," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013, pp. 1797–1800. DOI: `10.1109/ISCAS.2013.6572214`.

[61] D.-H. Le, T.-B.-T. Cao, K. Inoue, and C.-K. Pham, "A CAM-based Information Detection Hardware System for fast exact pattern matching," in *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2013, pp. 848–851. DOI: `10.1109/MWSCAS.2013.6674782`.

[62] T. Rao and T. Ikenaga, "Quadrant segmentation and ring-like searching based FPGA implementation of ORB matching system for Full-HD video," in *2017 Fifteenth IAPR International Conference on Machine Vision Applications (MVA)*, 2017, pp. 89–92. DOI: `10.23919/MVA.2017.7986797`.

[63] J. Huang, G. Zhou, X. Zhou, and R. Zhang, "A New FPGA Architecture of FAST and BRIEF Algorithm for On-Board Corner Detection and Matching," *Sensors*, vol. 18, no. 4, p. 1014, 2018. DOI: `10.3390/s18041014`.

[64] Q. Ni, F. Wang, Z. Zhao, and P. Gao, "FPGA-based Binocular Image Feature Extraction and Matching System," in *Proceedings of the 2019 4th International Conference on multimedia systems and signal processing*, ser. ICMSSP 2019, ACM, 2019, pp. 182–187, ISBN: 145037171X.

[65] Z. Peng, J. Wu, Y. Zhang, and X. Lin, "A high-speed feature matching method of high-resolution aerial images," *Journal of real-time image processing*, vol. 18, no. 3, pp. 705–722, 2020, ISSN: 1861-8200.

[66] T. HU and T. IKENAGA, "Pixel Selection and Intensity Directed Symmetry for High Frame Rate and Ultra-Low Delay Matching System," *IEICE transactions on information and systems*, vol. E101.D, no. 5, pp. 1260–1269, 2018, ISSN: 0916-8532.

[67] J. S. Lee, J. Yoon, and W. Y. Choi, "In-Memory Nearest Neighbor Search With Nanoelectromechanical Ternary Content-Addressable Memory," *IEEE Electron Device Letters*, vol. 43, no. 1, pp. 154–157, 2022. DOI: `10.1109/LED.2021.3131184`.

[68] A. Kazemi, M. M. Sharifi, A. F. Laguna, *et al.*, "FeFET Multi-Bit Content-Addressable Memories for In-Memory Nearest Neighbor Search," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2565–2576, 2022. DOI: 10.1109/TC.2021.3136576.

[69] E. Garzón, R. Golman, Z. Jahshan, *et al.*, "Hamming Distance Tolerant Content-Addressable Memory (HD-CAM) for DNA Classification," *IEEE Access*, vol. 10, pp. 28 080–28 093, 2022. DOI: 10.1109/ACCESS.2022.3158305.

[70] M. Irfan, A. I. Sanka, Z. Ullah, and R. C. Cheung, "Reconfigurable content-addressable memory (CAM) on FPGAs: A tutorial and survey," *Future Generation Computer Systems*, vol. 128, pp. 451–465, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.09.037. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21003836.

[71] Z. Ullah, M. K. Jaiswal, R. C. Cheung, and H. K. So, "UE-TCAM: An ultra efficient SRAM-based TCAM," in *TENCON 2015 - 2015 IEEE Region 10 Conference*, 2015, pp. 1–6. DOI: 10.1109/TENCON.2015.7372837.

[72] M. Irfan, Z. Ullah, M. H. Chowdhury, and R. C. C. Cheung, "RPE-TCAM: Reconfigurable Power-Efficient Ternary Content-Addressable Memory on FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 8, pp. 1925–1929, 2020. DOI: 10.1109/TVLSI.2020.2993168.

[73] M. Irfan, Z. Ullah, A. I. Sanka, and R. C. C. Cheung, "Accelerated Updating Mechanisms for FPGA-Based Ternary Content-Addressable Memory," *IEEE Embedded Systems Letters*, vol. 13, no. 2, pp. 37–40, 2021. DOI: 10.1109/LES.2020.2999471.

[74] M. Irfan, H. E. Yantır, Z. Ullah, and R. C. C. Cheung, "Comp-TCAM: An Adaptable Composite Ternary Content-Addressable Memory on FPGAs," *IEEE Embedded Systems Letters*, vol. 14, no. 2, pp. 63–66, 2022. DOI: 10.1109/LES.2021.3124747.

[75] I. Ullah, J.-S. Yang, and J. Chung, "ER-TCAM: A Soft-Error-Resilient SRAM-Based Ternary Content-Addressable Memory for FPGAs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 1084–1088, 2020. DOI: 10.1109/TVLSI.2020.2968365.

[76] 2005. [Online]. Available: http://pascal.inrialpes.fr/data/human/.

[77] C. G. Blair and N. M. Robertson, "Video Anomaly Detection in Real Time on a Power-Aware Heterogeneous Platform," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 11, pp. 2109–2122, 2016. DOI: 10.1109/TCSVT.2015.2492838.

[78] R. Dupre and V. Argyriou, "3D Voxel HOG and Risk Estimation," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, 2015, pp. 482–486. DOI: 10.1109/ICDSP.2015.7251919.

[79] *Xilinx technical documents, "PG151 - Divider Generator v5.1 Product Guide (v5.1)"*, 2016.

[80] L. Kalms, K. Mohamed, and D. Göhringer, "Accelerated Embedded AKAZE Feature Detection Algorithm on FPGA," in *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, ser. HEART2017, Bochum, Germany: Association for Computing Machinery, 2017, ISBN: 9781450353168. DOI: 10.1145/3120895.3120898. [Online]. Available: https://doi.org/10.1145/3120895.3120898.

[81] *G. Bradski, The OpenCV Library, Dr. Dobb's Journal of Software Tools*, OpenCV, 2000.

[82] V. Balntas, K. Lenc, A. Vedaldi, T. Tuytelaars, J. Matas, and K. Mikolajczyk, "H-Patches: A Benchmark and Evaluation of Handcrafted and Learned Local Descriptors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 11, pp. 2825–2841, 2020. DOI: 10.1109/TPAMI.2019.2915233.

[83] M. Riazi, M. Samragh, and F. Koushanfar, "CAMsure: Secure Content-Addressable Memory for Approximate Search," *ACM transactions on embedded computing systems*, vol. 16, no. 5s, pp. 1–20, 2017, ISSN: 1539-9087.

[84] K. Mueller and G. Trommer, "Real-time image matching and tracking for autonomous quadrotor helicopters," in *2019 26th Saint Petersburg International Conference on Integrated Navigation Systems (ICINS)*, 2019, pp. 1–8. DOI: 10.23919/ICINS.2019.8769431.

[85] M. Ali, A. Agrawal, and K. Roy, "Ramann: In-sram differentiable memory computations for memory-augmented neural networks," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '20, Boston, Massachusetts: Association for Computing Machinery, 2020, pp. 61–66, ISBN: 9781450370530. DOI: 10.1145/3370748.3406574. [Online]. Available: https://doi.org/10.1145/3370748.3406574.

[86] M. M. A. Taha and C. Teuscher, "Approximate memristive in-memory hamming distance circuit," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 2, Mar. 2020, ISSN: 1550-4832. DOI: 10.1145/3371391. [Online]. Available: https://doi.org/10.1145/3371391.

[87] *Kcu105 board user guide (v1.10) (ug917)*, UG917, v1.10, Xilinx, 2019. [Online]. Available: http://www.xilinx.com.

[88] H. Yang, G. Cheng, and H. Chen, "High Efficient Local Feature Matching," in *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*, 2018, pp. 2552–2556. DOI: 10.1109/IMCEC.2018.8469593.

[89] Y. Wang, X. Yang, X. Wang, C. Ke, and Q. Wang, "Application of improved SURF algorithm in real scene matching and recognition," in *2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL)*, 2020, pp. 536–541. DOI: 10.1109/CVIDL51233.2020.00-32.

[90] M.-W. Cao, L. Li, W.-J. Xie, *et al.*, "Parallel K Nearest Neighbor Matching for 3D Reconstruction," *IEEE Access*, vol. 7, pp. 55 248–55 260, 2019. DOI: 10.1109/ACCESS.2019.2912647.