Improving Image Matching using an Ensemble of Local Descriptors and Hardware Design

by

Sina Ghaffari
B.Sc., University of Tehran, 2015
M.Sc., Amirkabir University of Technology, 2017

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

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Sina Ghaffari, 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.

Improving Image Matching using an Ensemble of Local Descriptors and Hardware Design

by

Sina Ghaffari
B.Sc., University of Tehran, 2015
M.Sc., Amirkabir University of Technology, 2017

Supervisory Committee

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

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

Dr. Ulrike Stege, Outside Member
(Department of Computer Science)

Supervisory Committee

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

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

Dr. Ulrike Stege, Outside Member
(Department of Computer Science)

ABSTRACT

Image matching is one of the fundamental problems in computer vision, and has many applications such as object recognition, structure from motion, and 3D reconstruction. In this work, we aim to accelerate image matching algorithms, and improve their accuracy by proposing approaches that have a minimal impact on speed. With this in mind, we focus on handcrafted descriptor and matching algorithms. Our contributions in this dissertation are twofold.

The first set of contributions are related to the acceleration of descriptor algorithms and the reduction of resource utilization for image matching by proposing novel circuits based on Field Programmable Gate Arrays (FPGAs). We use FPGAs as a platform due to their features such as parallel processing, low-power computing, and flexibility in design.

This work presents a comprehensive analysis of FPGA-based implementations of the Histogram of Oriented Gradients (HOG) algorithm. A novel hardware-software co-design of the HOG algorithm is introduced to accelerate the execution of this descriptor algorithm. We propose methods such as logarithm-based bin assignment, approximate normalization, and a time-sharing protocol for sequential histogram generation for increasing the speed. A novel task allocation to optimize resource utilization on the hardware platform, in addition to acceleration of the HOG algorithm, is also presented.

Next, we focus on binary descriptors and present a novel hardware implementation of the Binary Robust Invariant Scalable Keypoints (BRISK) algorithm. BRISK is faster than non-binary descriptors but is computationally expensive with respect to other binary descriptor algorithms. A new sampling pattern for the BRISK algorithm is proposed to facilitate the hardware implementation of BRISK in multiple scales. Our proposed design reduces FPGA resource utilization while maintaining the image matching accuracy. Furthermore, the proposed fully pipelined design achieves a frame rate of 78 fps on images with full HD resolution.

The second set of contributions is related to improving image matching accuracy while maintaining performance in terms of computations. For this purpose, the focus is on handcrafted descriptor algorithms which are known to be more computationally efficient than deep learning based algorithms. We analyze

and propose fusion of descriptor algorithms which extracts complementary information to attain higher accuracy. To this end, three ensemble methods are proposed. The first method (weighted-fusion) combines a non-binary and a binary descriptor using their weighted distance metrics. The second method (binary fusion) combines a non-binary and a binary descriptor by converting the non-binary descriptor to a binary descriptor using a learned threshold. The third method (non-binary fusion) combines a non-binary and a binary descriptor by transforming the binary descriptor to a non-binary descriptor using a learned scaling factor. Comprehensive experiments on benchmarks from the HPatches, Brown (Photo tourism) and Oxford Affine Covariant Regions datasets are provided. The experimental results and analysis demonstrate a higher mean Average Precision (mAP) of the fusion methods in comparison with the baseline algorithms.

The next contribution for accuracy improvement is adding convolutional neural network (CNN) prefiltering to images prior to keypoint detection. The addition of a shallow CNN as the first step of a handcrafted algorithm to improve accuracy is proposed. The CNN is trained to filter the raw input images to achieve higher mAP. Experimental results indicate an improvement of accuracy using this method on the HPatches dataset.

Finally, we demonstrate our proposed approaches on a practical application. This application is relevant to environmental research by providing the basis for a tool for the automated identification of wildlife in tracking habitat (in this case, badgers). The proposed methods outperform the commonly-used handcrafted algorithms on identifying individual badgers in multiple images using their facial characteristics. This is done without fine-tuning the algorithms on the target badger identification dataset, which shows the generality of our proposed methods.

Contents

Supervisory Committee ii

Abstract iii

Contents v

List of Tables ix

List of Figures xi

Glossary xv

Acknowledgements xvi

1 Introduction 1

1.1 Motivation . . . 1
1.2 Image matching . . . 2
1.2.1 Applications of image matching . . . 4
1.3 Feature extraction and description . . . 7
1.3.1 Dense descriptors . . . 7
1.3.2 Local non-binary descriptors . . . 7
1.3.3 Local binary descriptors . . . 8
1.3.4 Learning-based descriptors . . . 8
1.3.5 Comparison of various categories of descriptor algorithms . . . 9
1.4 Ensemble of local descriptor algorithms . . . 11
1.5 Hardware acceleration of descriptor algorithms . . . 12
1.6 List of contributions . . . 12
1.7 Acknowledgment of author's contributions . . . 14
1.8 Structure of this dissertation . . . 14

2 Analysis and Comparison of FPGA-Based Histogram of Oriented Gradients Implementations 16

2.1 Introduction . . . 16
2.2 The HOG algorithm . . . 19
2.3 Optimizing the computation of the algorithm . . . 22
2.3.1 Input selection . . . 23

2.3.2 Magnitude calculation . . . 23
2.3.3 Orientation and bin assignment . . . 26
2.3.4 Normalization . . . 28
2.4 Data manipulation techniques . . . 28
2.4.1 Numerical representation . . . 29
2.4.2 Data flow modification . . . 29
2.4.3 Memory optimization . . . 30
2.5 Modified HOG-based features . . . 31
2.5.1 Feature compression . . . 31
2.5.2 Correlation of HOG features . . . 32
2.5.3 Histogram of significant gradients . . . 32
2.5.4 Gradient extension . . . 32
2.6 Hardware-software implementations . . . 33
2.7 Discussion . . . 34
2.7.1 Speed comparison . . . 34
2.7.2 Resource utilization . . . 36
2.7.3 Design guidelines . . . 36
2.8 Conclusion . . . 37

3 A Novel Hardware–Software Co-Design and Implementation of the HOG Algorithm 40

3.1 Introduction . . . 40
3.2 Review of the algorithm . . . 42
3.2.1 The HOG algorithm . . . 42
3.2.2 Support vector machine . . . 43
3.3 Related work on hardware–software implementations . . . 44
3.3.1 Hardware implementation of the HOG algorithm . . . 44
3.3.2 Hardware–software implementation of the HOG algorithm . . . 45
3.4 A novel hardware-software co-design of the HOG-SVM system . . . 46
3.5 HOG-SVM core . . . 48
3.5.1 Deserializer and buffer validity check . . . 48
3.5.2 Gradient and magnitude calculation . . . 49
3.5.3 Logarithm-based bin assignment . . . 49
3.5.4 One-row histogram generator . . . 52
3.5.5 One-cell histogram buffers . . . 53
3.5.6 Two-row histogram buffers . . . 53
3.5.7 Block normalization . . . 54
3.5.8 SVM classifier . . . 55
3.6 Results and comparison with other work . . . 56
3.7 Conclusions . . . 61

4 A Fully Pipelined FPGA Architecture for Multiscale BRISK Descriptors With a Novel Hardware-Aware Sampling Pattern 63

4.1 Introduction . . . 63
4.1.1 Hardware implementation of descriptors . . . 64
4.1.2 BRISK algorithm . . . 64
4.1.3 New approaches to enhance acceleration . . . 66
4.1.4 Organization of the paper . . . 66
4.2 Related work . . . 67
4.2.1 A comparison of binary descriptor algorithms . . . 67
4.2.2 FPGA-based implementations of binary descriptors . . . 68
4.2.3 FPGA-based implementations of BRISK descriptor . . . 68
4.3 Hardware-aware sampling pattern for BRISK . . . 69
4.4 Hardware implementation of the multi-scale BRISK algorithm . . . 70
4.4.1 BRISK pipeline . . . 74
4.4.2 Multi-scale BRISK . . . 76
4.5 Results and discussion . . . 77
4.5.1 Assessment of the new sampling pattern . . . 77
4.5.2 Effect of clock gating . . . 79
4.5.3 Sharing the multiplication stage . . . 79
4.5.4 Implementation results . . . 80
4.5.5 Accuracy evaluation . . . 88
4.5.6 Discussion . . . 89
4.6 Conclusion . . . 89

5 Learned Fusion of Complementary Local Descriptors for Improved Image Matching Accuracy 91

5.1 Introduction . . . 91
5.2 Related work . . . 94
5.2.1 Combining descriptors . . . 94
5.2.2 A brief review of the SIFT and BRISK algorithms . . . 95
5.3 Fusing binary and non-binary descriptors . . . 96
5.3.1 Weighted-fusion . . . 96
5.3.2 Binary-fusion . . . 97
5.3.3 Non-binary-fusion . . . 98
5.4 Experiments and results . . . 99
5.4.1 Learning approaches . . . 100
5.4.2 Experimental results and measurements on HPatches dataset . . . 100
5.4.3 Experimental results and measurements on Brown dataset . . . 104
5.4.4 Experimental results and measurements on Oxford dataset . . . 106
5.4.5 Discussion . . . 106
5.5 Conclusion . . . 109

6 Improving Handcrafted Image Matching Accuracy Using Learned Convolutional Filters and Processing Speed by Hardware Design 110

6.1 Introduction . . . 110
6.2 Related work . . . 111
6.2.1 Learning-based methods for image matching . . . 111
6.2.2 Convolutional neural networks as learned image filters . . . 112
6.2.3 Hardware implementation of image matching algorithms . . . 113
6.3 Learning-based convolutional filters . . . 113
6.3.1 Training using a hill climbing algorithm . . . 114
6.3.2 Parameter selection . . . 115
6.3.3 Baseline comparison . . . 117
6.4 Hardware implementation of prefiltered fusion algorithms . . . 118
6.4.1 Hardware evaluation . . . 122
6.5 Application to badger identification . . . 125
6.6 Conclusion . . . 129

7 Conclusions **130**
7.1 Contributions . . . 130
7.2 Future work . . . 132

References **134**

A Publications Arising from This Dissertation **148**

List of Tables

Table 1.1 Examples of commonly used descriptors . . . 10
Table 1.2 Estimation of the number of required multiplications and additions for L2-Net . . . 11

Table 2.1 Applications of FPGA-based HOG in the surveyed references . . . 17
Table 2.2 Table of notations . . . 18
Table 2.3 Operations required for the HOG algorithm for an M×N image . . . 21
Table 2.4 Speed comparison and FPGA used in each reference . . . 35
Table 2.5 Resource utilization . . . 37
Table 2.6 Optimization for speed . . . 37
Table 2.7 Optimization for accuracy . . . 38
Table 2.8 Optimization for hardware resource utilization . . . 38

Table 3.1 Limits for bin assignment . . . 51
Table 3.2 Block normalization decoding method . . . 55
Table 3.3 Comparison with other work . . . 58
Table 3.4 HOG-SVM IP-core resources . . . 59
Table 3.5 Resource usage of the whole hardware-software system . . . 60

Table 4.1 A comparison of examples of binary descriptors . . . 67
Table 4.2 Effect of changing the distance threshold T on the number of sample points with n=36 (θ=10) . . . 69
Table 4.3 Effect of changing the number of rotations on the number of sample points with T=1 70
Table 4.4 Approximations of multiplications in the filter unit . . . 73
Table 4.5 Adder tree number of additions and approximations . . . 76
Table 4.6 Number of required registers for pipeline implementation of the BRISK algorithm in original BRISK and our proposed pattern . . . 78
Table 4.7 Power consumption improvement using clock gating . . . 79
Table 4.8 Total resource usage for two scales of BRISK descriptor on KCU105 FPGA board . . 81
Table 4.9 Detail of resource usage in various steps of the BRISK descriptor . . . 82
Table 4.10 A summary of design metrics for recent FPGA implementations of binary descriptors 84
Table 4.11 Comparison of our design and other work in speed metric . . . 86
Table 4.12 Direct comparison of the sampling patterns . . . 89

Table 5.1 Summary of the three proposed methods . . . 93
Table 5.2 mAP measurement results of local descriptor algorithms with M=1 . . . 102

Table 5.3 mAP measurement results of local descriptor algorithms with M=3 102

Table 5.4 mAP measurement results of local descriptor algorithms with M=5 103

Table 5.5 False positive rate at 95% recall on Brown dataset 107

Table 6.1 Overall resource utilization of the proposed architecture on Kintex® Ultrascale™ FPGA (XCKU040) [109] 123

Table 6.2 Resource utilization of the modules of the proposed architecture 123

Table 6.3 Comparison of hardware implementation metrics for the SIFT descriptor with other work 124

Table 6.4 Comparison of our methods with other work 127

List of Figures

Figure 1.1 An example of keypoints in an image (left) and an extracted patch around the keypoint (right). . . . 3
Figure 1.2 The pipeline of an image matching system 3
Figure 1.3 An example of object recognition using image matching. The image is from HPatches dataset [11]. . . . 5
Figure 1.4 An example of image stitching application. The image is from the HPatches dataset [11]. . . . 6
Figure 1.5 Categorization of descriptor algorithms and example algorithms of each category . . . 9
Figure 1.6 Comparison of various types of descriptor algorithms in terms of speed, accuracy, and number of computations 10
Figure 1.7 The trade-offs among CPU, GPU, FPGAs, and ASICs platfroms 13

Figure 2.1 Categorization of HOG algorithm performance enhancement. 19
Figure 2.2 A flowchart of the HOG algorithm. The HOG algorithm is sequential, but the second step (magnitude and orientation) can be computed in parallel. 20
Figure 2.3 Visualization of cell and block in the HOG algorithm. 21
Figure 2.4 The flowchart of calculating a histogram of a cell from raw pixels. 22
Figure 2.5 A basic hardware architecture for magnitude approximation by the summation of absolute values of gradients in horizontal and vertical directions. 24
Figure 2.6 A basic hardware architecture for magnitude approximation. 24
Figure 2.7 A basic hardware architecture for magnitude approximation. 25
Figure 2.8 A basic hardware architecture for bin assignment. 26
Figure 2.9 A pipeline of three rows of line-buffer registers [80][63]. 30
Figure 2.10 Sliding window proposed by Qasaimeh et al. [65]. 31
Figure 2.11 The extension of HOG features used by Sledevič et al. [51]. 33

Figure 3.1 The KCU105 FPGA board (left) connected to the computer station (right) for experiments. 42
Figure 3.2 A flowchart of the HOG algorithm from input image sensor to HOG features. 42
Figure 3.3 Visualization of cells (4 by 4 pixels) and blocks (each containing 4 cells) in the HOG algorithm. 43
Figure 3.4 Block diagram of the proposed design and port connections 47
Figure 3.5 The overall diagram of the HOG-SVM core. 48
Figure 3.6 Line buffers in the deserializer module. 49

Figure 3.7 Difference between the slope of three logarithm functions. 50
Figure 3.8 The bin assignment procedure. 51
Figure 3.9 One-row histogram generation module. 52
Figure 3.10 The time-sharing protocol for the histogram generation module. 53
Figure 3.11 The block diagram of one-cell histogram buffers. 54
Figure 3.12 The block diagram of two-row histogram buffers. 54
Figure 3.13 The block diagram of block normalization module. 55
Figure 3.14 The SVM classifier module. 56
Figure 3.15 SVM block internal logic. 57
Figure 3.16 The relation between pixel stride and frame rate. 59
Figure 3.17 Comparison of software implementation and our proposed HW-SW co-design. 61

Figure 4.1 Original BRISK sample points [22] (©[2011] IEEE). 65
Figure 4.2 Examples of long pairs (left image) and short pairs (right image) based on the BRISK sample pattern [22] (©[2011] IEEE). 65
Figure 4.3 Image matching pipeline. 68
Figure 4.4 Examples selected from the sequence of steps for generating the proposed sampling pattern (with T=1, n=36). 70
Figure 4.5 Examples of sampling patterns resulting from changing threshold (T) to produce different sampling patterns for the same number of rotations n. 70
Figure 4.6 Examples of sampling patterns resulting from changing the number of rotations n to produce different sampling patterns for the same threshold T. 71
Figure 4.7 The overall architecture of our design. 71
Figure 4.8 Architecture of the FAST keypoint detector. 72
Figure 4.9 Weights of the filter unit that is shown in Fig. 4.7. 73
Figure 4.10 The block diagram of a pipeline architecture for one scale. 74
Figure 4.11 Two parallel 10-level adder trees for two channels. 75
Figure 4.12 Short pair computation step. 77
Figure 4.13 Shared multiplication logic between two scales. 80
Figure 4.14 Timing of the valid data on the 1st and 2nd scales. 80
Figure 4.15 Matching results using recall vs. 1−precision curve for Boat, Wall, and Graffiti image sets of the Oxford Affine Covariant Regions dataset [111]. 86
Figure 4.16 Matching results using recall vs. 1−precision curve for Leuven, Trees, and UBC image sets of the Oxford Affine Covariant Regions dataset [111]. 87
Figure 4.17 Comparison of the original BRISK algorithm and HWBRISK$_{final}$ in different scales using mean AUC (area under curve) for images from the Oxford Affine Covariant Regions dataset [111]. 87

Figure 5.1 Categorization of local descriptor algorithms into non-binary and binary, and hand-crafted and learning-based algorithms. 92
Figure 5.2 Comparison between descriptor vector generation of SIFT and BRISK. 96

Figure 5.3 Block diagram of weighted-fusion for fusion of a non-binary and a binary descriptor algorithms using distance metric. 96
Figure 5.4 Block diagram of binary-fusion for fusion of a binary and a converted non-binary descriptors. 98
Figure 5.5 Block diagram of non-binary-fusion for fusion of a converted binary and a non-binary descriptors. 98
Figure 5.6 Comparison of mAP measurement results of our proposed methods with SIFT and BRISK descriptor algorithms. The threshold distance value for evaluation is $M = 1$. . . 102
Figure 5.7 Comparison of mAP measurement results of our proposed methods with SIFT and BRISK local descriptor algorithms. The threshold distance value for evaluation is $M = 3$. 103
Figure 5.8 Comparison of mAP measurement results of our proposed methods with SIFT and BRISK local descriptor algorithms. The threshold distance value for evaluation is $M = 5$. 103
Figure 5.9 The visualization of 500 matches with the lowest computed distances in pairs of images from (a) Castle, Bark, Greenhouse, Astronautis and (b) Home, Artisans, Woman, and Calder image sets from the HPatches dataset [11]. 105
Figure 5.10 Experimental results on patch matching benchmark of the HPatches dataset. 106
Figure 5.11 Recognition rate comparison of our proposed methods with LMBD on four image sets of the Oxford Affine Covariant Regions dataset [111]. 108

Figure 6.1 The block diagram of our proposed method. 114
Figure 6.2 An example of the generation of the learning mask (a) and an example of the generation of the new filter solution based on the previous filter and the learning mask (b). 115
Figure 6.3 Architecture of our proposed shallow CNN and an example of the input image and filtered image. The example image is selected from the Apprentices image set of the HPatches dataset [11]. 116
Figure 6.4 Loss curve over training iterations for various learning-rate (α) values. We choose $\alpha = 0.2$ for other experiments since the loss value does not have a noticeable change with $\alpha > 0.2$. 116
Figure 6.5 The mAP curves by changing the number of (a) layers n_l, and (b) sublayers n_{sl} of the CNN model. 117
Figure 6.6 The comparison of SIFT and conv-SIFT algorithms using mAP on 5-fold cross validation on the HPatches dataset [11]. 118
Figure 6.7 The mean Average Precision comparison on 5-fold cross validation on the HPatches dataset [11] for an acceptance error margin of 3. 118
Figure 6.8 Example of matches detected in HPatches images using SIFT and conv-SIFT. 119
Figure 6.9 Functional block diagram of the hardware architecture 120
Figure 6.10 The overall architecture of our proposed hardware accelerator. 121
Figure 6.11 Steps of the SIFT descriptor generation. 122
Figure 6.12 The block diagram of the normalization step for the SIFT descriptor vector. 124

Figure 6.13 Heatmap of confusion matrix. 126
Figure 6.14 Examples of matches found on badger facial images. 128

Glossary

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
BELID	Boosted Efficient Local Image Descriptor
BEBLID	Boosted Efficient Binary Local Image Descriptor
CORDIC	Coordinate Rotation Digital Computer
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
FAST	Features from Accelerated Segment Test
FPGA	Field-Programmable Gate Arrays
FREAK	Fast Retina Key-point
GPU	Graphics Processing Unit
HOG	Histogram of Oriented Gradients
LMBD	Learning Multiple local Binary Descriptor
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 everyone who has supported me throughout my PhD journey and contributed to the successful completion of this dissertation.

I would like to express my deep appreciation to my two supervisors, Dr. Kin Fun Li and Dr. David Capson, for their invaluable guidance, support, and feedback. Throughout my PhD journey, I have learned a lot from them, not only in terms of my research but also in terms of professional development. I am grateful for their support and for the opportunities they have provided me to grow both academically and professionally.

I would like to give special recognition to my wife, Parastoo Soleimani, without whom this dissertation would not have been possible. I know that with you by my side, anything is possible. I am looking forward to taking our next steps together and I am excited about what the future holds for us.

I am also grateful to my parents, Farshideh Ghandi and Abbas Ghaffari, who supported me in my path leading to my PhD journey. I would also like to acknowledge my colleagues and friends who have supported me throughout my PhD journey.

Finally, I would like to express my gratitude to the University of Victoria for providing me with the opportunity to pursue my PhD and for the support and resources that have been made available to me. I am grateful for the education and experiences that I have gained during my time here.

Thank you all for being a part of my journey and for your contributions to the completion of this dissertation.

Chapter 1

Introduction

1.1 Motivation

Designing computer algorithms for creating the ability to see and make decisions has been a long-lasting goal for many researchers in computer vision community in the last decades. Computer vision algorithms are proposed to enable the computers to see, understand the images, and make decisions. Some of the fundamental problems in computer vision, which are a base for higher level applications, include object detection, recognition, and finding similarities among objects in different images. Although there has been many achievements in some of these computer vision applications, one of the major obstacles of using computer vision in practical applications is the processing speed.

The main motivation behind this research work is increasing the abilities of computer vision algorithms to be used in practical applications that can facilitate our lives. This motivation leads us to focus on enhancing the implementation of practical computer vision applications that are currently limited due to their processing time and accuracy.

In support of our motivation, our goal in this research is to improve the performance of image matching algorithms, which are the base for other practical algorithms, to attain more reliable outputs in terms of accuracy while maintaining acceptable processing speed. This goal can be achieved by focusing on design ideas for increasing the overall accuracy of computer vision applications without sacrificing speed.

Another motivation of this work is to demonstrate the potential of our algorithms to applications with environmental benefits. To this end, the proposed algorithms in this work are applied to the application of badger identification as a case study.

This dissertation presents our research towards the goals described in this section as well as ideas, experiments, and evaluation results.

In this chapter, first, we introduce the image matching problem, the steps of image matching pipeline, and its applications in section 1.2. Then, we present a more detailed introduction on feature extraction and description in section 1.3. After that, we introduce the ensemble of local descriptor algorithms for improving image matching accuracy in section 1.4. In section 1.5, we discuss the hardware implementations of descriptor algorithms for acceleration. We present a list of our contributions in this work in section 1.6. In section 1.7 we provide information and acknowledgment of author's contribution in jointly authored works related to this dissertation. Finally, we present the structure of the rest of this dissertation in section 1.8.

1.2 Image matching

Image matching is one of the fundamental problems in computer vision and a base for many applications such as image registration [1], visual SLAM (Simultaneous Localization and Mapping) [2], and structure from motion [3]. The main goal of the image matching algorithms is to find the correspondences of an object of interest or a scene in two or multiple images. An image matching system comprises a number of steps including scale-space generation, keypoint detection, patch description, keypoint matching, and outliers removal, where each step can be implemented using a variety of algorithms. There has been much work in the literature in recent years focusing on the improvement of accuracy and the speed of image matching algorithms [4]–[7]. Improving accuracy makes the algorithms more robust in challenging practical applications and the focus on speed is due to the high computational and real-time requirements of the algorithms used in image matching.

In this section, the image matching pipeline is introduced. The goal of image matching algorithms is to pair an object or a scene in two or multiple images from different viewpoints. Some of the challenges in image matching include different illumination conditions and occlusion by other objects which are present in the images. In addition, some images are captured from a variety of angles which result in different viewpoints of an object. The image matching process is a sequential pipeline of algorithms which consists of finding keypoints in two images, extracting the descriptor vectors around the keypoints, and matching the set of descriptor vectors from the first image to the second image. To better explain the concept of image matching, we define some important terminology in the image matching literature as follows:

keypoint keypoints, also called interest points, are noticeable pixels in an image or video which have some variations with respect to the surrounding pixels. keypoints present changes in the color or texture of an image. As an example, the corners or the edges of an object are typically considered as keypoints. In image matching applications, we use a variety of algorithms such as Harris [8], Hessian [9], or FAST [10], to detect keypoints in images.

Patch Patches are small windows of pixels in an image. In image matching algorithms, we usually extract patches around a keypoint (the keypoint is the center of the patch) and use the information extracted from the pixels in the patch to describe the local area around that keypoint. An image patch can have various sizes depending on the algorithm. Figure 1.1 illustrates an example of keypoints in an image and a patch extracted from that image.

Local descriptor There are many algorithms for extracting features in a local patch around a keypoint. These algorithms are called local descriptors since they focus on a relatively small and local part of an image. The goal of a local descriptor in image matching is to generate feature vectors (descriptor vectors) around the keypoints which are similar, for the same physical points of an object as seen in two different images. At the same time, the descriptor vectors should not be similar for keypoints that do not correspond to the same physical points. The similarity of two descriptor vectors is measured using standard distance functions such as Euclidean, Hamming, or cosine distances. The choice of a distance function is based on the descriptor algorithm.

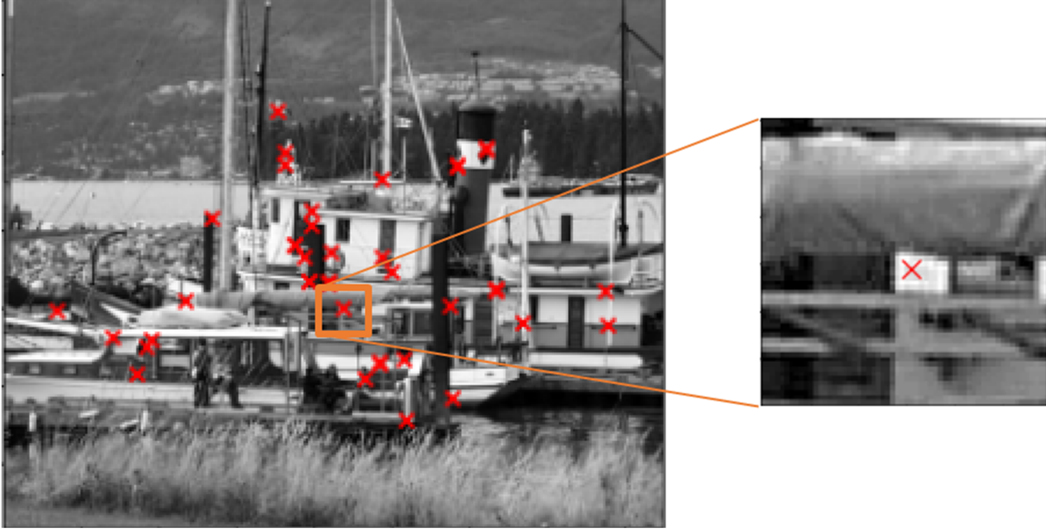

Figure 1.1: The original image is from the HPatches dataset [11].

Since image matching comprises several steps, we can model the image matching problem as a pipeline of steps where each step can be processed by various algorithms. The image matching pipeline is shown in Fig. 1.2. The first step of image matching is scale space generation. In this step, various sizes of the image are created so that the algorithm can find the keypoints of the objects in images with different scales. The second step is keypoint detection. In this step, the image is scanned using a keypoint detection algorithm and the coordinates of the keypoints are extracted. After that, the third step is patch description. In this step, a descriptor vector is generated by extracting features from the pixels in a local patch around the keypoint. The step after patch description is the keypoint matching step. In this step, the descriptor vectors extracted from the first image are compared to the descriptor vectors of the second image using a distance metric. The goal of this step is to find a set of pairs from the descriptor vectors of the two images that have the least distances. The final step of the image matching pipeline is outlier elimination (or outlier removal). In this step, the detected matches and the physical coordinates of the keypoints are examined to reduce the number of false positive matches.

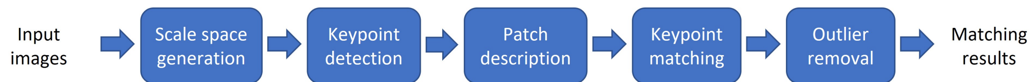

Figure 1.2: The pipeline of an image matching system

Although good performance of any of the steps in the image matching pipeline is essential, improving the patch description step can have significant effects on accuracy. In addition, the most computationally expensive step in most algorithms is patch description which is the bottleneck of the image matching pipeline. As a result, the main focus of this dissertation is on the patch description stage.

1.2.1 Applications of image matching

Image matching has many applications in various fields of computer vision. In this section, we review some examples of applications of image matching including object recognition, visual SLAM, structure from motion, and image stitching. These applications require a high number of computations in the image matching stage. Increasing the speed of computations make the corresponding algorithms more suitable for real-time systems. As an example, visual SLAM, which is a robotic application, is typically implemented on embedded systems with real-time constraints. Therefore, increasing the speed of the algorithms for these applications while maintaining the accuracy is essential.

Object recognition

Object recognition is the application of locating an object of interest in an image. In this problem, there is usually a reference image which contains the object of interest. The goal is to locate the same object in a different (target) image. The object of interest can have a different size in the target image than the reference image. It can also have rotation variation. In some applications, the object of interest might be occluded by other objects in the scene. There are many algorithms and various methods for object recognition in the literature, such as CNN and HOG-SVM classification. One of the well-known methods for object recognition is image matching. In this method, the keypoints of the object, such as its corners or edges, are detected in both the reference image and the target image. Then, the keypoints are matched together using the description of the patches around them. If there is a consensus among the direction of the matches and the location of the keypoints, the location of the object of interest is deemed to be found in the target image.

Figure 1.3 shows an example of object recognition using image matching techniques. In Fig. 1.3, the left image is the reference image which shows the object of interest, and the image on the right is the target image and the keypoints are indicated using red circles. The green lines demonstrate the correspondences of the keypoints which are matched using the SIFT local description algorithm [12]. Although there are other methods for object recognition in the literature, using local descriptor algorithms has the advantage of being more robust to occlusion. The weakness of image matching for object recognition is that the presence of repeatable textures in the object of interest can lead to a poor matching result.

Structure from motion

Structure from motion (SfM) is a technique to generate three dimensional models from a sequence of 2 dimensional images based on motion changes and the camera positions. SfM can be used in many practical applications such as augmented reality, hand-eye calibration, motion capture, and image-based 3D modeling [3]. In SfM applications, image matching is used to find correspondences among images and based on the relative distance of the correspondences, the depths of the image points are calculated. Finally, by having the depth information, a 3D model is constructed from the sequence of images. Structure from motion algorithms are computationally expensive both in the image matching step and in the 3D reconstruction step. Accelerating the image matching step, the first step of SfM, can lead to an overall speed up of the SfM process. In addition, by producing more accurate matches in the first step, the accuracy of the 3D model also increases.

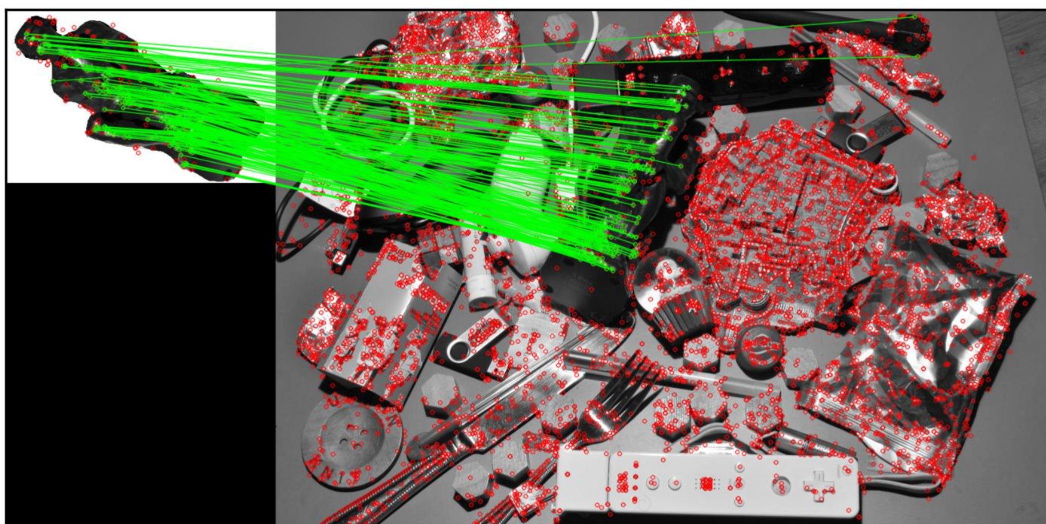

Figure 1.3: The red circles represent the keypoints and the green lines represent the matchings found between the two images

Visual SLAM

Simultaneous location and mapping (SLAM) is a robotic application that benefits from image matching [2]. One branch of the family of SLAM algorithms is visual SLAM which processes input images. In a visual SLAM application, the keypoints from consecutive frames of a video are matched. Based on the motion vectors between each two consecutive frames, the camera movement is estimated. In the final stage, the view of the camera is matched with the first frame (or the starting point of movement). By using a visual SLAM algorithm, one can simultaneously locate a moving agent in an environment while also generate a map of the unknown environment from the input frames. This application has essential value in the environments where GPS signals are not easily accessible. In many applications, visual SLAM is implemented on embedded systems and the real-time implementation of this algorithm is challenging. By utilizing the benefits of parallel processing and hardware acceleration, other more robust algorithms can also be used for this application.

Image stitching

Another application of image matching is image stitching [13]. Image stitching is the process of combining multiple images which have overlaps in certain regions of the same scene or object. In this application, first, the interest points in two (or more) images are detected using a detector algorithm. Then, local descriptor vectors are extracted from the patches around each keypoint. After matching the descriptor vectors, a unified rotation and scale transformation is estimated for the viewpoint of each of the images with respect to the reference image. Then, by transforming the images using the obtained parameters, the images are combined into a single image. Figure 1.4 shows an example of image stitching. Some of the difficulties in image stitching are related to the quality and various conditions in which the images are taken. For example, lens distortion, scene motion, and light exposure can produce challenges in image stitching. Therefore, the local descriptor and image matching algorithms used in image stitching should be robust to these types of variations to produce more accurate results.

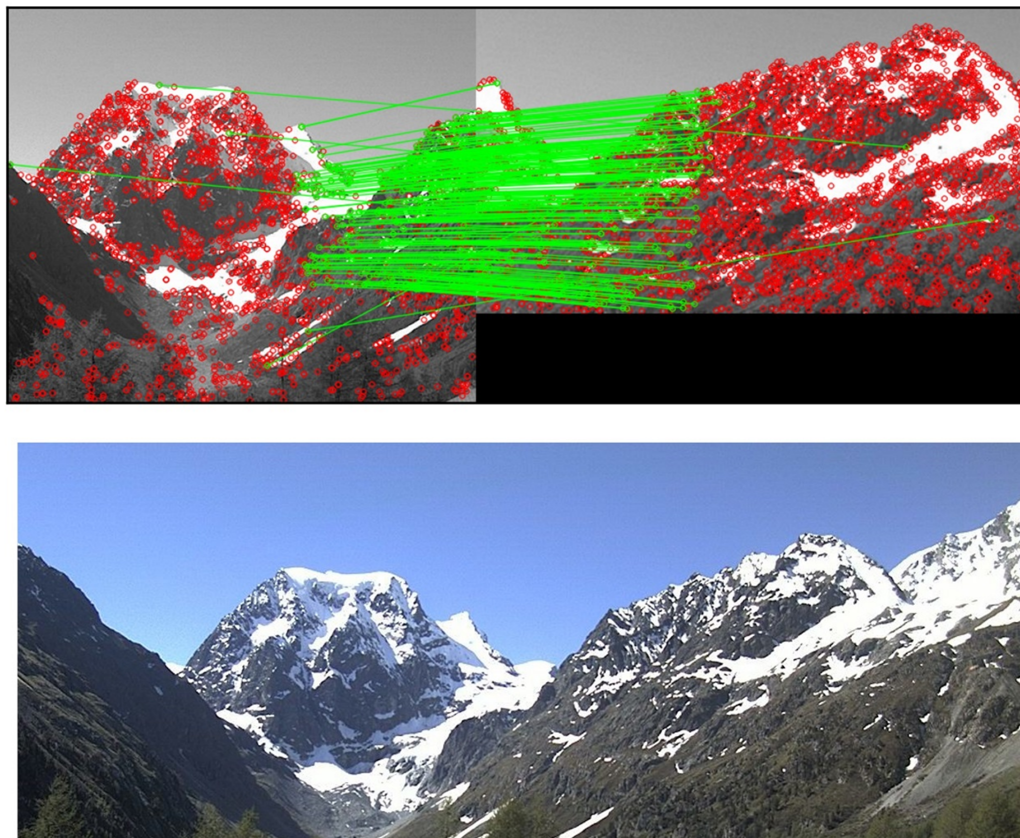

Figure 1.4: An example of image stitching application. The image is from the HPatches dataset [11].

Image matching is a fundamental problem in computer vision with numerous applications [4]. In this section, four examples of image matching in a wide range of applications were introduced. However, there are some challenges such as computational complexity and the high number of computations required in using image matching algorithms in practical situations. These challenges can result in non-real-time execution. There have been many attempts to improve the speed of these algorithms in recent years. However, increasing the speed of the algorithms and improving the accuracy for practical applications remain as challenging tasks in the computer vision community. The patch description stage is a common step of the mentioned applications and one of the bottlenecks of image matching system in terms of speed. The performance of the description step directly affects the accuracy of the image matching systems. We provide a general discussion on description and feature extraction in section 1.3, and we focus on this stage in this research and propose methods for improving local descriptor algorithms in the following chapters. In section 1.4, the ensemble of local handcrafted descriptor algorithms is introduced as a method to improve the accuracy of image matching with minimal additional parameters.

1.3 Feature extraction and description

Since the majority of the contributions of this dissertation are related to improving the patch description stage of the image matching pipeline, we discuss description and feature extraction in this section in more detail. Feature extraction is one of the essential parts of pattern recognition and computer vision. Features are processed information extracted from the raw input data, such as images or frames of a video in computer vision. They usually have lower dimension than the initial input data and contain information that can be useful to make a decision about the raw input data. The extracted features are also called descriptor vectors and the feature extraction algorithms are called descriptor algorithms in computer vision literature. The descriptor algorithms can be categorized into multiple groups based on their design methodology, generated output, and their application.

In this section, we introduce some of the commonly known descriptor algorithms. We start by introducing dense descriptors, which extract features from every pixel of an image in 1.3.1. Then, we introduce local descriptors, which extract features from selected parts of an image in 1.3.2 and 1.3.3, and finally, an introduction on learning-based descriptors is presented in 1.3.4.

1.3.1 Dense descriptors

Dense descriptor algorithms extract features from every pixel (or small groups of pixels) in an image. One of the applications for dense descriptors is object detection [14]. One of the most popular algorithms used for dense description is the Histogram of Oriented Gradients (HOG) [15]. The HOG algorithm generates a descriptor vector for the image by computing local histograms based on the orientation of gradients for each pixel in the image. The HOG algorithm has been used in many applications. A disadvantage of the HOG algorithm in practical application is the complexity in computation and the high number of required computations which result in increased processing time. This disadvantage has been addressed by efficient algorithm-specific hardware implementations for the HOG algorithm. A literature review on proposed hardware implementations of the HOG algorithm to accelerate the HOG description is presented in chapter 2 of this dissertation. For the purpose of accelerating the HOG algorithm, a novel hardware-software co-design of the HOG algorithm is presented in chapter 3.

1.3.2 Local non-binary descriptors

Another category of the descriptor algorithms is local descriptors. Local descriptor algorithms extract features from local areas of the images which have a potential to present more useful information for specific applications. Some of the applications that benefit from local descriptor algorithms are image matching, image retrieval and image registration. For all these applications, first, a number of keypoints are detected in an image. keypoints are pixels in an image such as edges or corners which usually present a change in an image. The main goal for selecting a keypoint is for the algorithms to be able to find the same keypoint in another image of the same scene under various image transformations. After selecting the keypoints, a patch of pixels around a keypoint is selected and the descriptor algorithm extracts features (a.k.a. a descriptor vector) from the pixels surrounding that keypoint. The size of the image patches can be different based on the descriptor algorithm and the scale of the image which is being processed. Some evaluation benchmarks use 65×65 pixels while others use 33×33 pixels as the size of the image patch [11].

Local descriptor algorithms are categorized into binary descriptors and non-binary descriptors based on the generated output by the algorithm. The non-binary descriptors generate a vector of floating-point values (or 8-bit values after normalization) while the binary descriptors generate a vector of zeros and ones. Typically, non-binary descriptors require more computation and result in higher accuracy while the binary descriptors are faster in execution and have lower memory footprint.

There are many local non-binary descriptor algorithms in the literature such as SURF[16], SGLOH [17], and L2-Net [18]. One of the most popular local non-binary descriptors is the Scale Invariant Feature Transform (SIFT) [19] algorithm. The SIFT algorithm is very similar to the HOG algorithm as it also computes the histogram of oriented gradients. However, SIFT has some additional steps such as scale-space generation, keypoint detection, and main orientation estimation. The difference of SIFT and HOG in the description stage is that SIFT just computes the HOG vectors in a patch of pixels around a keypoint, while HOG computes the vectors densely in an image. Our methods for improving the accuracy of image matching by using the SIFT algorithm are presented in chapters 5 and 6. Similar to HOG, the SIFT algorithm also suffers from large numbers of computations and complex mathematical operations. We present a hardware design for efficient implementation and acceleration of the SIFT algorithm in chapter 6.

1.3.3 Local binary descriptors

Local binary descriptors are proposed to enhance the speed of the non-binary descriptor algorithms. BRIEF [20], ORB [12], and FREAK [21] are some of the commonly used local binary descriptor algorithms found in the literature. Binary descriptor algorithms usually use a comparison test in the last stage to generate a binary vector. The comparison test compares the intensity of predefined sample pixels in an image patch surrounding the detected keypoints and based on the result of comparison, assigns a value of 0 or 1 to the elements of the descriptor vector.

One of the popular local binary descriptors in computer vision literature is the Binary Robust Invariant Scalable Keypoints (BRISK) [22]. BRISK has shown superior results in terms of accuracy with respect to other binary descriptors in some applications [23]. Similar to the SIFT algorithm as discussed in section 1.3.2, the BRISK algorithm has multiple steps for patch description. The orientation assignment step of the BRISK algorithm requires a high number of computations including multiplications and additions. In order to improve the speed of the BRISK algorithm for practical applications with hard time constraints, we present a multi-scale FPGA-based implementation of the BRISK algorithm in chapter 4.

1.3.4 Learning-based descriptors

Another categorization of descriptor algorithms is to divide them into handcrafted and learning-based algorithms. Handcrafted algorithms are designed based on image features and are typically more generic and useful for a diverse range of applications. On the other hand, learning-based algorithms learn a number of parameters by training over a data set to enhance the accuracy of the learned descriptor model. The number of parameters to be learned varies depending on the descriptor algorithm and can be in a range of a few parameters to millions of parameters (in deep learning algorithms). The deep learning based descriptor algorithms such as L2-Net [18] and HardNet [24] use convolutional neural networks to extract features from image patches. Although the methods based on deep learning achieve state-of-the-art results in terms of accuracy in applications such as image matching, they require a large number of computations and bigger memory

storage in comparison with other methods [25]. The required processing time of deep learning based descriptors are usually higher and not comparable with hand-crafted methods or simpler learning-based models when implemented on the same platform.

Some of the other learning-based methods which do not use deep convolutional neural networks are BEBLID [26], BinBoost [27], and LMBD [28]. Our method for learning the parameters of a descriptor model to improve handcrafted non-learning based descriptor algorithms is presented in chapter 5.

1.3.5 Comparison of various categories of descriptor algorithms

Figure 1.5 illustrates the categorization of descriptor algorithms and gives examples of each category. Figure 1.6 illustrates an intuitive relation of non-binary, binary, and learning-based descriptor algorithms. Deep learning based descriptor algorithms obtain the best results in terms of accuracy. However, they are computationally expensive with respect to other types of descriptor algorithms. On the other hand, binary handcrafted descriptors are usually the faster algorithms but they cannot obtain comparable accuracy results as non-binary hand-crafted descriptor algorithms. Other learning-based descriptors such as BEBLID [26] and Binboost [27] are more similar to handcrafted binary or non-binary descriptor algorithms in terms of accuracy.

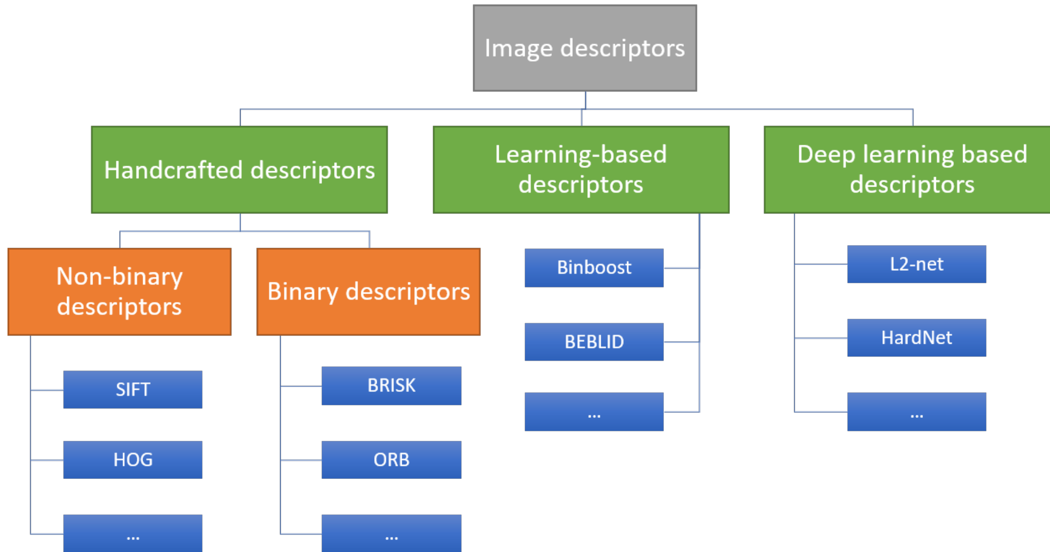

Figure 1.5: Categorization of descriptor algorithms and example algorithms of each category

Some examples of the commonly used local descriptor algorithms and the type of their output feature vectors are shown in Table 1.1.

One disadvantage of deep learning-based descriptors is the required number of operations. As an example, L2-Net [18], which is a commonly used architecture in deep learning based descriptors, requires about 133 million multiplications and 125 million additions to compute one descriptor vector. This value is not comparable with the lower amount of computations required by handcrafted descriptors. The number of multiplications in each layer of a deep neural network presented by [18] is shown in Table 1.2. The number of

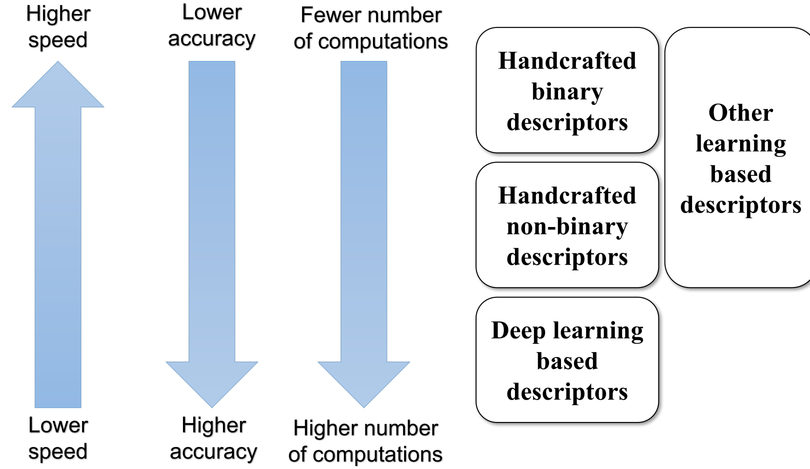

Figure 1.6: Comparison of various types of descriptor algorithms in terms of speed, accuracy, and number of computations

Table 1.1: Examples of commonly used descriptors

Algorithm	Type	Descriptor vector type and dimension
SIFT [19]	Handcrafted non-binary	128 floating-point
SURF [16]	Handcrafted non-binary	64 floating-point
HOG [15]	Handcrafted non-binary	Dense descriptor
BRIEF [20]	Handcrafted binary	512 bits
BRISK [22]	Handcrafted binary	512 bits
FREAK [21]	Handcrafted binary	512 bits
ORB [12]	Handcrafted binary	256 bits
AKAZE [29]	Handcrafted binary	486 bits
BinBoost [27]	Learning-based	256 bits
BELID [30]	Learning-based	512 floating-point
BEBLID [26]	Learning-based	256 bits
LMBD [28]	Learning-based	256 bits
L2-Net [18]	Deep learning-based	256 floating-point
HardNet [24]	Deep learning-based	128 floating-point
TCDesc [31]	Deep learning-based	128 floating-point

multiplication operations for each layer of a convolutional neural network is computed using equation (1.1):

$$\text{Number of multiplication operations} = \frac{k_1 \times k_2 \times n_1 \times n_2 \times W \times H}{s^2} \tag{1.1}$$

where k_1 and k_2 are the dimensions of the kernel of the current layer, n_1 is the depth of the current layer and n_2 is the depth of the next layer, s is the stride, W is the width of the current layer and H is the height of the current layer. If the stride is equal to 2, W and H are divided by the stride value. The number of addition operations is computed as shown in equation (1.2):

$$\text{Number of addition operations} = \frac{(k_1 \times k_2 - 1) \times n_1 \times n_2 \times W \times H}{s^2} \tag{1.2}$$

In comparison, binary descriptor algorithms require fewer computations than deep learning based de-

Table 1.2: Estimation of the number of required multiplications and additions for L2-Net

Layer	Input size	Convolution parameters	Stride	Multiplications	Additions
1	32×32×1	3×3×1	1	295k	262k
2	32×32×32	3×3×32	1	18.8M	16.8M
3	32×32×64	3×3×64	2	9.4M	8.4M
4	16×16×64	3×3×64	1	18.8M	16.8M
5	16×16×128	3×3×128	2	9.4M	8.4M
6	8×8×128	3×3×128	1	9.4M	8.4M
7	8×8×128	8×8×128	1	67M	66M

scriptor algorithms. As an example, the BRISK algorithm processes a 35×35 patch for each keypoint. For each patch, 876 long pairs and 512 short pairs of pixels are selected. For orientation assignment, BRISK requires 876 subtractions, 1752 multiplications and summations, and one arc tangent operation. For description, only 512 comparisons are required. Other non-binary or binary handcrafted algorithms have differing numbers of operations. However, the BRISK example shows that the amount of computations required by deep learning based algorithms is not comparable with a typical binary descriptor algorithm.

1.4 Ensemble of local descriptor algorithms

There have been many local descriptor algorithms proposed to achieve higher accuracy in recent years. Many of the descriptor algorithms extract complementary information and features from the image patches. As an example, some algorithms such as SIFT and SURF extract gradient information from local dense pixels in a patch while other algorithms such as ORB and BRISK apply binary comparison test on predefined sample locations around the keypoints for patch description. The difference in the extracted features brings out the opportunity to combine the descriptor algorithms in an ensemble model and utilize the various types of distinctiveness offered by multiple algorithms to obtain better results.

Ensemble methods are used in machine learning applications to improve the accuracy of the algorithms. In ensemble methods, a set of models or algorithms are combined and applied on the same input data. Based on the usage of the same base models or different base models, the ensemble can be homogeneous or heterogeneous, respectively. There are several methods for combining the machine learning models. The most well-known methods are bagging, boosting, and stacking [32].

Boosting methods are sequential in learning. In boosting methods, each model uses the samples that were classified incorrectly by the previous models to improve the accuracy. Bagging and stacking models are learned on the dataset in parallel. Bagging methods usually contain homogeneous models and learn independently from each other. Stacking methods are similar to the bagging methods in terms of learning the models independently and in parallel. However, the base models are different in stacking methods.

One of the advantages of using stacking ensemble methods on local descriptors is that the models can learn complementary features from a dataset. Descriptor algorithms such as BRIEF, BRISK, and ORB extract statistical information from the difference of local pixels in a patch. On the other hand, descriptor algorithms such as SIFT and SURF extract information from smaller windows of pixels. Even among the binary descriptors, some methods such as AKAZE generate the descriptor vector based on information from rectangular samples while others use a one-pixel binary comparison tests for descriptor generation.

In this dissertation, various methods for using a stacking ensemble of local descriptor algorithms are

presented to achieve higher accuracy. If the descriptor vectors of multiple algorithms for each image patch are generated in parallel, the overall speed of the image matching pipeline is not reduced. To this end, we discuss hardware acceleration of descriptor algorithms which also supports parallel implementation in section 1.5.

1.5 Hardware acceleration of descriptor algorithms

Due to the complexities and high number of computations of computer vision algorithms, hardware platforms are getting attention for implementation. The most commonly-used hardware platforms in computer vision are central processing units (CPU), graphics processing units (GPU), field programmable gate arrays (FPGAs), and application-specific integrated circuits (ASIC). Figure 1.7 illustrates the differences among these platforms in terms of flexibility, speed and cost per unit. Although GPUs offer some level of parallel processing for simple calculations, FPGAs and ASICs can be used to implement algorithm-specific hardware which is more efficient than CPUs and GPUs in terms of speed and power consumption. On the other hand, implementing an algorithm for CPUs requires less time and effort. In this dissertation, we focus on FPGA platforms as they offer opportunities to implement image matching algorithms using specific designed hardware while having more flexibility than ASIC platforms.

In recent years, there have been much work focusing on FPGA-based implementation of the different steps in image matching algorithms [33]–[35]. The advantage of using such platforms is the possibility of parallel computation and utilization of algorithm-specific circuits. Since the processing of an image is usually based on multiple repeating operations on each of the pixels, many image processing algorithms, including local descriptor algorithms, can benefit from parallel processing at the pixel level. In this way, the algorithm is not limited to sequential execution which is conventional in general CPU units. On the other hand, designing a circuit specific to local descriptor algorithms can result in a noticeable speed up of the process as shown by Ulusel et al. [36] for the BRIEF and BRISK algorithms. In this dissertation, we demonstrate improvement in the speed of the image matching pipeline using hardware acceleration. Having the hardware architecture requirements in mind, we propose methods to modify the algorithms to maintain and improve accuracy as well.

1.6 List of contributions

In this chapter, the image matching concept was introduced and a few examples of image matching applications were presented. In addition, different categorizations of local descriptor algorithms (binary/non-binary, handcrafted/learning-based, local/dense) were reviewed. In this section, an overview of the contributions of this dissertation is presented.

- A survey and a comprehensive analysis on hardware acceleration of a dense non-binary descriptor algorithm (HOG) [37], [38]: The methods and techniques used for hardware acceleration of the histogram of oriented gradients algorithm (HOG) are surveyed in detail and presented in chapter 2. In addition, a comprehensive comparison of the recent methods and guidelines for design choices based on the requirements of applications are presented which can be used by other researchers.

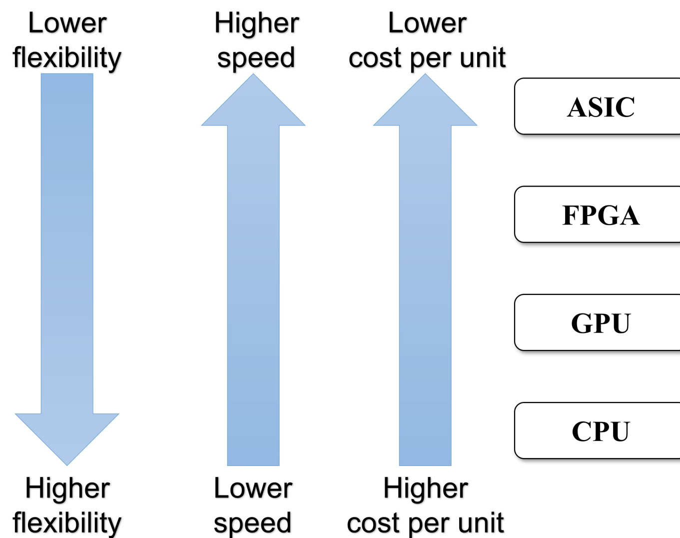

Figure 1.7: The trade-offs among CPU, GPU, FPGAs, and ASICs platfroms

- Hardware acceleration of a non-binary dense descriptor algorithm (HOG) [39]: A novel hardware-software co-design of the HOG algorithm which increases the speed of this algorithm is presented in chapter 3. In this design, a logarithm-based bin assignment technique is introduced which reduces hardware resource utilization for FPGA implementation of the HOG algorithm. In addition, an efficient task allocation method for implementation of the HOG algorithm is proposed. The proposed design achieves a frame rate of 115 frames per second while using less hardware resources in comparison with other existing work.

- Hardware acceleration of a binary descriptor algorithm (BRISK) [40]: a novel hardware design for implementation of a multi-scale BRISK algorithm is proposed. In this work, a modified hardware-efficient sampling pattern for the BRISK algorithm is introduced which leads to a more efficient implementation of this algorithm in multiple scales in terms of speed. The proposed design which is presented in chapter 4 achieves a frame rate of 78 frames per second on images with full HD resolution.

- Learning-based fusion of a non-binary and a binary descriptor with complementary information [41]: Three learning-based methods of weighted-fusion, binary-fusion, and non-binary fusion are proposed and analyzed for the combination of a binary and a non-binary descriptor algorithms. As a case study, the experimental results on fusion of the SIFT and BRISK descriptor algorithms are analyzed on multiple benchmarks and presented in chapter 5.

- A training procedure for a shallow convolutional neural network as the first step of image matching pipeline for improving accuracy [42]: A method based on a hill climbing algorithm for learning filters from a training data set is presented in chapter 6 for preprocessing of the images so that the accuracy of image matching using handcrafted methods is improved.

- An overall hardware architecture for the fusion of the BRISK and SIFT algorithms [42]: A unified hardware architecture for acceleration of the SIFT and BRISK algorithms is proposed. The overall

architecture includes convolutional filtering method as well and is presented in chapter 6.

- An application for demonstrating the advantages of our proposed methods [42]: Applying the proposed generalized fusion of local descriptors and convolutional prefiltering to the badger identification application as a case study of a difficult matching problem. The results and analysis of our proposed methods are presented in chapter 6.

Our contributions in this dissertation include a set of methodologies which pave the way for many other future research directions. Our contributions to hardware implementation are not limited to the HOG and BRISK algorithms. The proposed novel sampling pattern and the contributions to the BRISK and the HOG architecture can be applied to other binary and non-binary descriptor implementations. Our proposed methods can be applied to a wide variety of applications since the result of image matching is an initial input for many other algorithms.

1.7 Acknowledgment of author's contributions

This section provides an acknowledgement of author's contributions to joint authorship under two supervisors. The presented work in [38] analyzes various FPGA-based implementations of the HOG algorithm. In [38], the FPGA-based implementations of the HOG algorithm are divided into four categories. The first category focuses on the analysis of various methods for optimization of the computation of each individual step of the HOG algorithm. The second category discusses data manipulation techniques such as numerical representation, data flow modification, and memory optimization. In the third category, newly introduced modified HOG-based features with FPGA implementation are discussed, and the hardware-software co-design solutions are presented in the fourth category. The contributions of the author of this dissertation includes the survey and analysis of the first and second categories, the guideline design tables, and the overall comparison of all work provided in [38]. The third and fourth categories presented in [38] are the contributions by the second author.

Our work in [39] comprises a novel hardware-software co-design of the HOG algorithm. This work presents four contributions. The first contribution is a new task allocation of the HOG descriptor algorithm to hardware and software platforms. The second contribution is the logarithm-based bin assignment and the third and fourth contributions are simplification in the normalization step and parallel histogram generation, respectively. The contributions of the author of this dissertation comprises the first, third, and fourth contributions as well as the implementation of the software code and the interface to the hardware circuit. Coding and implementation of the HOG core and the second contribution (logarithm-based bin assignment) are the contributions of the other authors of [39].

1.8 Structure of this dissertation

In this chapter, the problem of image matching and image description in computer vision was introduced. Various categorizations of image descriptor algorithms were discussed and some example applications of image matching were presented. In addition, the advantages of implementing descriptor algorithms on hardware platforms were discussed. In the remainder of this dissertation, the work that has been published

and under preparation resulting from the research work is presented. This dissertation follows a publication-based approach in which each of the chapters 2, 3, 4, 5, and 6 is also a stand-alone publication. As a result, some necessary review is also provided in the beginning of each chapter.

In chapter 2, we present our publication on the analysis and comparison of different FPGA based methods for the implementation of the HOG algorithm [37], [38]. The HOG algorithm is a dense descriptor algorithm which requires complex operations.

Chapter 3 presents our work on a hardware-software co-design of the HOG algorithm on an FPGA platform [39]. In chapter 3, we also present a case study of human detection using the HOG algorithm.

Chapter 4 focuses on a hardware implementation for acceleration of the BRISK algorithm which is a local binary descriptor algorithm [40]. In chapter 4, a hardware architecture for the implementation of multi-scale BRISK algorithm is also presented in detail.

In chapter 5, an analysis of three learning-based fusion methods of a binary and a non-binary local descriptor algorithms is presented [41]. Multiple benchmarks and evaluation settings are provided to present the advantages of these proposed methods.

Chapter 6 provides the results of adding shallow convolutional network as a prefilter to the image matching pipeline [42]. In addition, an overall hardware architecture which combines the BRISK and SIFT algorithm with convolutional prefiltering is presented. Since the fundamental stages of the SIFT descriptor algorithm are based on the HOG algorithm, the overall hardware design presented in chapter 6 combines the contributions presented in chapters 3 and 4. Furthermore, the results of applying our proposed techniques on the badger identification problem is discussed in chapter 6.

In chapter 7, the research work discussed in this dissertation is concluded and potential future paths for continuing this research are suggested.

Chapter 2

Analysis and Comparison of FPGA-Based Histogram of Oriented Gradients Implementations

One of the commonly-used feature extraction algorithms in computer vision is the histogram of oriented gradients. Extracting the features from an image using this algorithm requires a large amount of computations. One way to boost the speed is to implement this algorithm on field programmable gate arrays, to benefit from flexible designs such as parallel computing. In this paper, we first, provide a summary of the steps of the histogram of oriented gradients algorithm. We then survey the implementation techniques of the histogram of oriented gradients on field-programmable gate arrays in the past decade. We group the different techniques into four main categories and analyze various enhancement methods in each category. The first group is the optimization of the algorithm computation which involves the steps of input selection, magnitude calculation, orientation and bin assignment, and normalization. The second category is data manipulation techniques which include numerical representation, data flow modification, and memory optimization. The third group contains modified features based on the histogram of oriented gradients and their hardware implementation, and the fourth one is the implementations in hardware-software co-design of the algorithm. We compare the different implementations using a speed metric called pixels per clock cycle, and resource utilization. Finally, we provide design summary tables for efficient implementation with respect to the speed metric, accuracy, and resource utilization.

2.1 Introduction

One of the most well-known feature extraction algorithms in computer vision is the histogram of oriented gradients (HOG). Dalal and Trigs [15] present 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 [43][44], texture classification [45], vehicle detection [46], and human activity recognition

Table 2.1: Applications of FPGA-based HOG in the surveyed references

Application	FPGA-based HOG implementation projects
Pedestrian detection*	[53][54][55][52][56][57][58][59][60][61][62][63][64][65][66][67]
Human detection*	[48][14][68][69][50][70][71][72][73]
Car detection	[74][75][76]
Traffic sign detection	[77]
Crowd density estimation	[71]
General object detection	[49][78][79]
Object tracking	[80][51]
Feature matching	[65]
Anomaly detection	[81]
Digit recognition	[82]

* The most common applications are pedestrian and human detection.

[47]. A complete object detection model can be designed by using HOG features coupled with a classifier such as Adaboost [48] or Support Vector Machines [14][49]. Some work focus on modified extended features based on HOG descriptors [50][51].

Since many applications in computer vision have real-time constraints and are implemented as an embedded system, much research has been focusing on the hardware acceleration of computer vision algorithms. Although HOG has shown outstanding detection capacity, 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 such as Graphical Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs). Since many applications require mobility and low power consumption, the implementation of the HOG algorithm has been more popular using FPGAs than GPUs.

FPGA implementations can potentially run faster, with less resource utilization and power consumption, which are important in embedded systems that require real-time processing. Ma et al. [52] compare CPU, GPU and FPGA implementation of the HOG algorithm. They implement the HOG algorithm on an Intel Xeon E5520 CPU processor, an Nvidia Tesla K20 GPU and a Xilinx Virtex-6 FPGA. To process a single frame, their FPGA implementation consumes 130x less energy than that of the CPU and 31x less energy than that of the GPU, while the speed is about 68x faster than the CPU and 5x faster than the GPU.

Over the years, many researchers propose FPGA implementations for the HOG algorithm in many different applications. Table 2.1 shows the applications of the FPGA-based HOG algorithm reviewed in this paper.

One of the most popular applications which has used HOG features is pedestrian detection. Pedestrian detection and tracking have been employed in driving assistance, surveillance, and robotics. Other popular applications of the HOG algorithm are for human detection. The difference between pedestrian and human detection is that pedestrians are mostly standing and walking in different directions while in the case of human detection, people may be in any possible position such as playing a sport, dancing, or just stationary. Some papers demonstrate the HOG algorithm in traffic sign detection [77] and car detection [74], which are useful in autonomous vehicle systems. Blair et al. [81] propose an application of the HOG algorithm to locate illegally parked cars in urban areas. Other applications, as shown in Table 2.1, include crowd density

Table 2.2: Table of notations

Notation	Description
θ	Orientation
BRAM	Block RAM
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
DSP	Digital Signal Processor
FIFO	First In First Out
FIND	Feature Interaction Descriptor
FPGA	Field-Programmable Gate Arrays
Gx	Gradient in horizontal direction
Gy	Gradient in vertical direction
GPU	Graphics Processing Unit
h	Histogram vector
hn	Normalized histogram vector
HOG	Histogram of Oriented Gradients
HSG	Histogram of Significant Gradients
HW/SW	Hardware/Software
IoT	Internet of Things
IP	Intellectual Property
LUT	Look-Up Table
RGB	Red, Green, Blue color channels
SVM	Support Vector Machine
SoC	System on Chip

estimation, digit recognition, and general object detection.

This paper is an extended version of our previous work [37]. We review the research work on FPGA-based HOG implementations from 2010 to 2019, collected from IEEE Xplore, Science direct, and NCBI databases. The selected papers are the most relevant articles that we could identify for HOG implementation on FPGA platforms. Table 2.2 presents the notations and their description used in this paper.

The typical parameters in an embedded real-time FPGA-based system which could be optimized are speed, power, resource usage, and accuracy. There is always a trade-off between these parameters. For example, by parallelizing the design and using more FPGA resources, one can improve speed. Most of the papers in this survey focus on speed and resource utilization. Only a few of them have reported on power consumption while others have assumed that implementation on an FPGA consumes less power than GPUs and CPUs, and therefore is naturally of lower power.

In this survey, we discuss the different techniques and methods of implementing the HOG algorithm on FPGA. First, we review the HOG algorithm in section 2.2. We then group these methods into four main categories based on different techniques to enhance HOG implementation. Fig. 2.1 shows the organization of the various categories of the survey provided in this paper. In section 2.3, we present innovative methods which optimize the computation of different steps of the algorithm. In section 2.4, we discuss in detail the techniques in recent work which are related to data structure and manipulation. In section 2.5, we present the FPGA implementation of some modified versions of the HOG algorithm. In section 2.6, we review the methods which benefit from hardware-software co-design. Finally, we have a critique on these methods in section 2.7, and a comparison of the results of recent papers is presented.

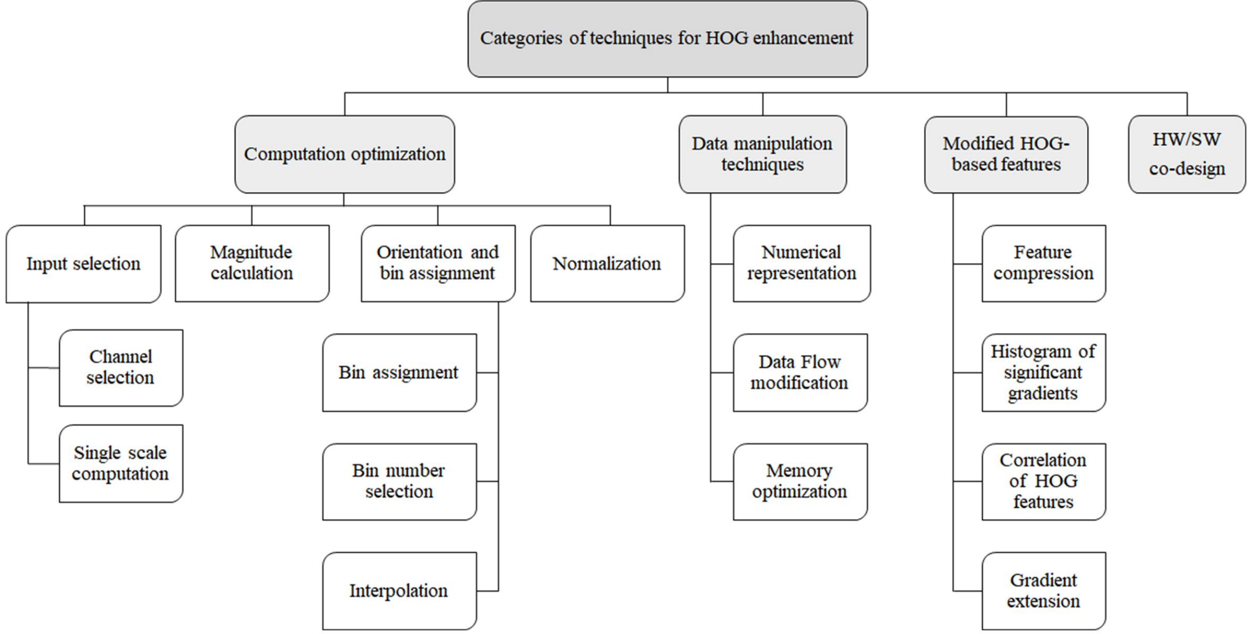

Figure 2.1: Categorization of HOG algorithm performance enhancement.

2.2 The HOG algorithm

The basic idea of the HOG algorithm is to use gradient information of each pixel to extract discriminating features for object detection. HOG features are normally extracted from various window sizes in the image. Fig. 2.2 shows a flowchart of this algorithm.

In the original HOG algorithm [15], the image window is divided into several blocks, and each block is divided into several cells. As an example, each block may contain four cells, and each cell may contain 16 (4×4) pixels. The first step of the HOG algorithm, as shown in Fig. 2.2, is to compute the gradients of each pixel in each cell, that is, to compute the derivatives in horizontal and vertical directions using the pixels around them. The gradient of the image is computed 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 $I(x,y)$ is the image pixel with coordinates x and y, G_x is the gradient of the horizontal direction and G_y is the gradient of the vertical direction. After calculating the gradients, the second step of the HOG algorithm is to compute the magnitude and orientation of each pixel as shown in (2.3) and (2.4):

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

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

The third step of the HOG algorithm is bin assignment in which a histogram is created based on the

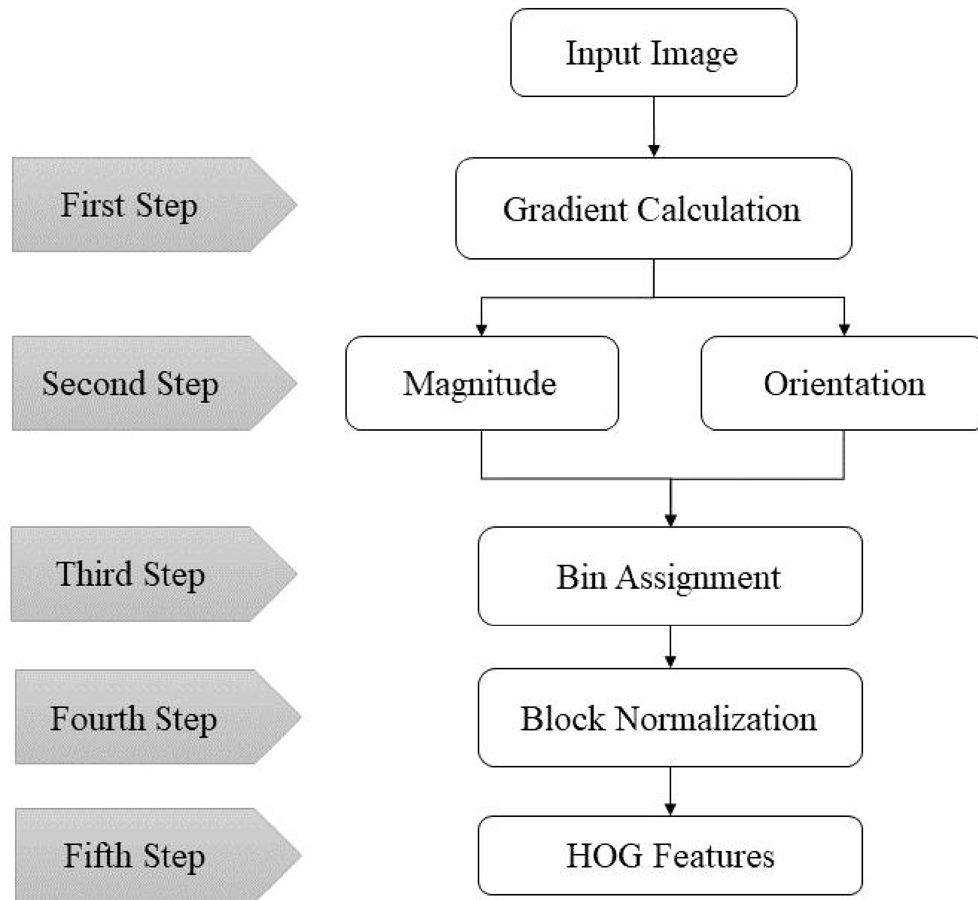

Figure 2.2: A flowchart of the HOG algorithm. The HOG algorithm is sequential, but the second step (magnitude and orientation) can be computed in parallel.

calculated orientation of pixels in each cell, which could be between 0 to 180, or 0 to 360 degrees, depending on the implementation configuration. Dalal and Trigs [15] use nine bins each corresponding to 20 degrees in their original work. The magnitude of each pixel is added to the value of the bin which contains the orientation of that pixel. In order to reduce aliasing, the weighted magnitude of each pixel is added to two adjacent bins based on the distance of its orientation to the center of the bins.

In the fourth step, the histograms of cells within each block are normalized separately. Finally, HOG features are obtained by concatenating all histogram values in the selected window in the fifth step. Fig. 2.3 shows how the input image window is divided for an example in which each cell contains 16 pixels, and each block contains four cells. The arrows in each pixel represent the orientations of the gradients in that cell, and a histogram is created for each cell.

The flowchart in Fig. 2.4 shows the steps of the HOG implementation for each cell of the image. After calculating the histograms for each cell, the cell histograms within a block are normalized together. Therefore, for this example, in each cell, 32 gradients, 16 magnitudes, and 16 orientation values are computed. Then, by comparing the orientations of the pixels with bin limits, the appropriate bin is assigned for all 16 pixels. After that, we have four histograms that are divided by the L2-norm of themselves.

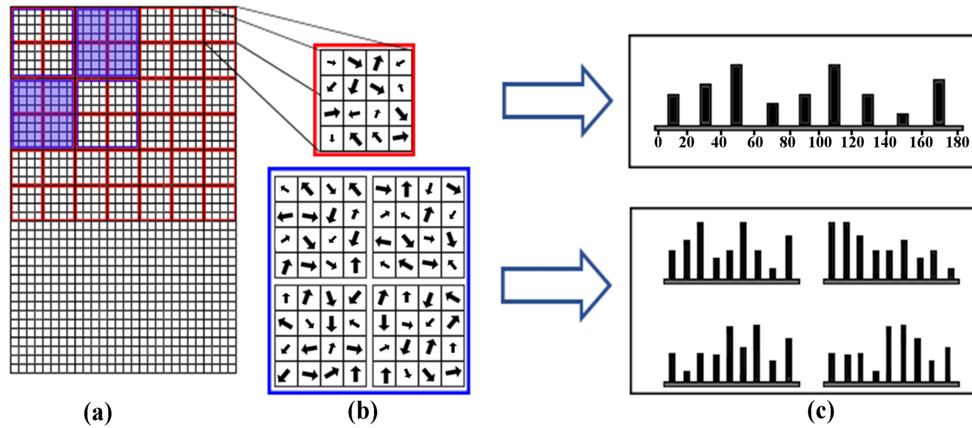

Figure 2.3: Visualization of cell and block in the HOG algorithm. (a) shows how pixels in an image window are grouped by cells and blocks. Each 4 by 4 pixels (red square) is a cell and each blue 8 by 8 pixels (blue square) is a block. The top part of (b) shows orientation of gradients in one cell and the bottom part of (b) shows the orientations in one block. The top part of (c) shows an example histogram for each bin created from one cell and the bottom part of (c) shows an example of 4 histograms in one block which are going to be concatenated and normalized in the next step. The number under each bin limit on the upper figure shows the ranges of the orientations of that bin.

Table 2.3: Operations required for the HOG algorithm for an M×N image

Stages of HOG	Multiplication	Addition	Square root	Arctangent	Comparison	Division
Gradient	$9MN$	$2MN$	—	—	—	—
Magnitude	$2MN$	MN	MN	—	—	—
Orientation	—	—	—	MN	—	MN
Bin assignment	—	—	—	—	$9MN$	—
Normalization	$9KB$	$9KB$	B	—	—	$9KB$
Overall	$11MN + 9KB$	$3MN + 9KB$	$MN + B$	MN	$9MN$	$MN + 9KB$

In general, for an M by N input image, the first step of the HOG algorithm requires $9 \times M \times N$ multiplications and 2xMxN additions to compute its gradient in x and y directions. In the second step, the algorithm computes the magnitude and arctangent for each pixel. Therefore, $M \times N \times 2$ multiplications, $M \times N$ additions, and $M \times N$ square root and arctangent evaluations are required. In the third step, the angle for each pixel is compared with the limit values between 0 to 180 (or -90 to 90) degrees. Depending on the algorithm and data, the number of comparisons in this step might differ. But for nine bins, the maximum number of comparisons is nine. Then, for each pixel, the magnitude value is added to a bin value. Therefore, there are $M \times N$ additions in this step. After that, in the normalization step, the histograms of a block are normalized. For K cells in each block, we have $9K$ divisions, $9K$ multiplications and additions, and one square root computation. The total number of computations is shown in Table 2.3.

Each line shows the required number of operations for each stage of the HOG algorithm. The last row presents the overall summation of required operations. The symbols represent: M = height of the image, N = Width of the image, K = number of cells in each block, B = total number of blocks in an image.

In Table 2.3, M and N are the dimensions of the input image, K is the number of cells in each block and B is the total number of blocks in the image. Table 2.3 is useful as it can show us by simplifying each stage

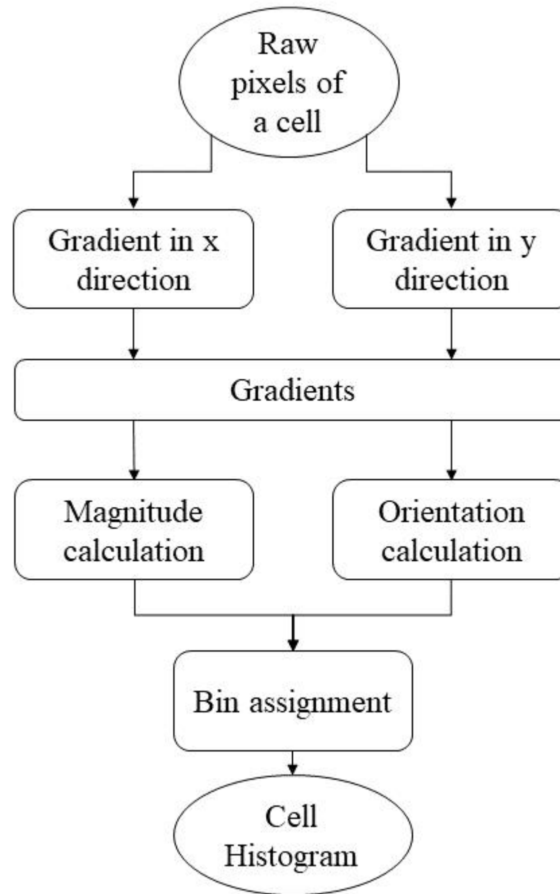

Figure 2.4: The flowchart of calculating a histogram of a cell from raw pixels. Parallel computation is possible for gradients in x and y directions and after that for magnitude and orientation calculation. The final result of this stage is a histogram of one cell.

of the algorithm, how many operations are affected. As an example, we can transform the multiplications in the gradient calculation step to subtractions and additions. Also, we have two steps containing square root. If we could optimize the calculation of square root, both these steps would benefit from it. For example, suppose that $M = N = 200$ and we have 16 pixels in each cell and 4 cells in each block. Therefore, K and B would be 2500 and 625, respectively. In this case, we have about 14M multiplications and additions, 40k square root calculations, 40k arctangent operations, 14M divisions, and 360k comparisons.

2.3 Optimizing the computation of the algorithm

The HOG algorithm contains several steps which are shown in the flowchart of Fig. 2.2. In this section, first, we categorize the different input selection choices in section 2.3.1. The methods which optimize magnitude calculation are reviewed in section 2.3.2. Then, since many work have integrated orientation calculation and bin assignment techniques, we review these two steps together in section 2.3.3. Finally, the normalization step is discussed in section 2.3.4.

2.3.1 Input selection

In this section, we review different color channels and inputs to the HOG algorithm.

Channel selection

Several papers implement the HOG algorithm on the luminance channel. Rettkowski et al. [60] propose to first convert the RGB image into a luminance image for hardware implementation. Then, they compute the gradients of the luminance image using line buffers. Advani et al. [50] propose an initial stage to find the dominant channel (from RGB) for HOG calculation. In this method, first, the gradients of each cell of the image for the three RGB channels are calculated and based on the accumulation of their values, a comparator chooses which channel is the most suitable for further processing. Ilas [76] simplifies the input channel even further by proposing to compute HOG on binary images instead of grayscale images. First, the image is transformed from a greyscale image to a binary image by comparing each pixel to a threshold. Then, the HOG features are extracted.

Observations and Conclusions: Performing HOG on a binary image reduces the delay and hardware utilization since only one bit is used for each input pixel. However, since there is less information embedded in the extracted features, the accuracy of the model decreases as well. This method can be useful if the speed and hardware utilization are very critical in the specific application and the contour of the object of interest in the image does not contain many details. Otherwise, the luminance channel is the most commonly used in HOG feature extraction.

Single scale computation

While some researchers such as Blair et al. [68], Ma et al. [52], and Li et al. [79] implement the HOG algorithm on multi-scale images, Negi et al. [48] simplify the algorithm and compute HOG on a single scale image. Ma et al. [52] work on 640×480 pixel frames and use 1.05 scaling factor for multi-scale detection with a window stride of four pixels. They employ 34 scales for HOG extraction and achieve 68.18 frames per second with 640x480 frames.

Observations and Conclusions: Multi-scale detection, which means using different sizes of the image to extract features, requires extra hardware resources for resizing the input image and buffering it inside the FPGA. It should only be used if the accuracy is application dependent or there are varying sizes of the object of interest in the image; otherwise, single-scale detection is faster than multi-scale detection.

2.3.2 Magnitude calculation

The next step in the HOG algorithm is to compute the gradients of each pixel. In order to do so, the difference of pixels in rows and columns are calculated. The magnitude of gradients is computed as shown in (2.3) in section 2.2. Since square and square root calculations are complex in hardware, several papers approximate the magnitude calculations. For FPGA implementation, in order to reduce the number of square root calculations in magnitude computations, Chen et al. [80] approximate the gradient amplitude for each pixel and simplify it according to (2.5) and (2.6):

$$Magnitude(x,y) = |I(x+1,y) - I(x-1,y)| + |I(x,y+1) - I(x,y-1)| \tag{2.5}$$

$$Magnitude(x, y) = |G(x)| + |G(y)| \quad (2.6)$$

Fig. 2.5 shows the block diagram of the hardware architecture described in (2.6).

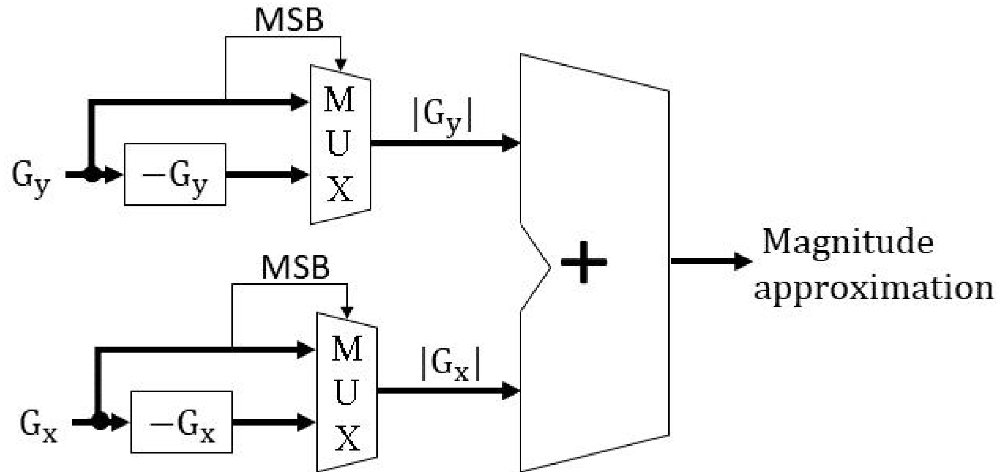

Figure 2.5: A basic hardware architecture for magnitude approximation by the summation of absolute values of gradients in horizontal and vertical directions. MSB is the most significant bit of the signal and the opposite sign is computed as two's complement. This architecture requires one adder, two multiplexers, and two two's complement operations.

Blair et al. [68] use the approximation mentioned by Wilson et al. [65], as shown in (2.7):

$$Magnitude = \frac{1}{1+\sqrt{2}}(|G_x| + |G_y| + \sqrt{2}max(|G_x|, |G_y|)) \quad (2.7)$$

The block diagram of the hardware architecture of (2.7) is shown in Fig. 2.6.

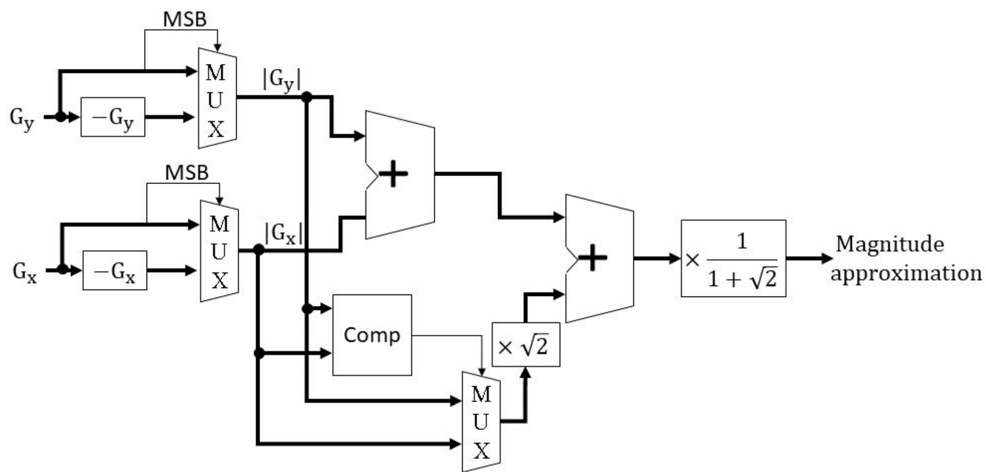

Figure 2.6: A basic hardware architecture for magnitude approximation. MSB is the most significant bit of the signal and the opposite is computed as two's complement. This architecture requires three multiplexers, two adders, two multipliers, one comparator (shown as Comp) and two two's complement operations.

Chen et al. [69], B.K. et al. [82], and Wang et al. [62] use the square root approximation technique [83], which approximates magnitude calculation using (2.8):

$$Magnitude(x, y) = max((0.875a + 0.5b), a) \tag{2.8}$$

where $a = max(G_x, G_y)$ and $b = min(G_x, G_y)$. In this way, the magnitude can be estimated using comparators and shifts only, thus simplifying the hardware. Fig. 2.7 presents the block diagram of the hardware design of (2.8).

Rettkowski et al. [60] store the magnitudes in look-up tables and compute the square root of magnitudes by retrieving the approximate values from the look-up tables. The total memory required for these look-up tables is 72 KB which can be stored in a block RAM in the FPGA.

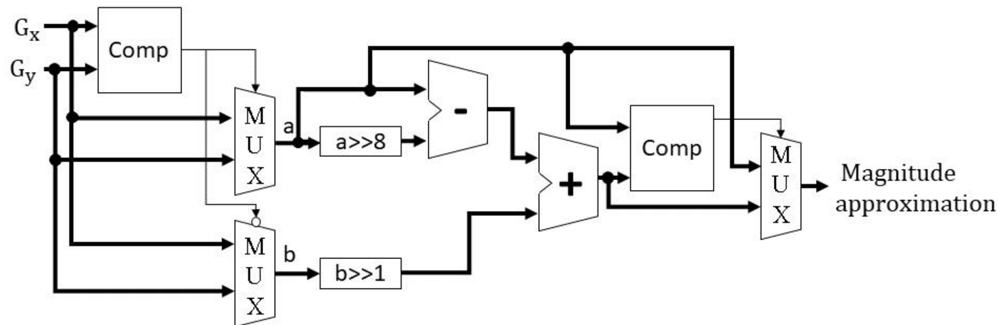

Figure 2.7: A basic hardware architecture for magnitude approximation. The ">> n" sign means signed shifting to the right by n bits. This architecture requires three multiplexers, one adder, one subtractor, two shifting operations, and two comparators (shown as Comp).

Some researchers use pre-developed IP cores for magnitude calculation in the HOG algorithm. Sledeviè et al. [51], Luo et al. [63], and Li et al. [79] use the Altera ALTSQRT IP core to calculate the square root function. Huang et al. [71] use a Xilinx IP core for square root calculation.

Observations and Conclusions: There are several ways to improve speed and hardware utilization of the magnitude computation. The simplest method, which is used by Chen et al. [80], uses only two two's complement units, two multiplexers, and one adder. More complex method, such as the work proposed by Blair et al. [68], which requires two multiplications, two additions, three multiplexers and one comparison, results in a more accurate approximation of the magnitude computation. On the other hand, the square root approximation technique consumes fewer hardware resources as it uses only shift, add and comparison. Since the magnitude should be computed for every pixel, it has a great effect on the overall speed of the circuit. Therefore, methods proposed by Chen et al. [69], B.K. et al. [82], and Wang et al. [62] are suitable for high speed requirements. If accuracy is more important than speed, using IP cores for exact computation leads to a higher accuracy. If hardware utilization is not a concern in the design, the method provided by Rettkowski et al. [60] can be fast and accurate as well.

2.3.3 Orientation and bin assignment

The next step of the HOG algorithm is computing the gradient orientation and assigning the magnitudes to the proper bins. Several papers make innovative contributions to this part of the algorithm.

Bin assignment

Many papers suggest that bin assignment can be performed without computing the value of arctangent for gradient orientation. Blair et al. [68] use the method described in the work provided by Bauer et al. [67] for assigning gradient magnitudes to bins. For orientation calculation, Blair et al. [68] use approximate values for tangent of the orientation (tan(θ)) and compare the Gy(x,y) (gradient in the vertical direction) value with tan(θ) multiplied by $G_x(x, y)$ (gradient in the horizontal direction). This method, which consumes less hardware resources than computing the tangent of the division of G_y over G_x for each value, is shown in (2.9).

$$G_x(x, y)tan(\theta_i) \leq G_y(x, y) < G_x(x, y)tan(\theta_{i+1}) \tag{2.9}$$

where θi is one of the limit angles and θi+1 is the limit after that. Hahnle et al. [53], Hemmati et al. [55], Chen et al. [69], Zhou et al. [77], Wang et al. [62], and Luo et al. [63] implement bin assignment in the same way. By using this method, they decrease the amount of hardware required for their HOG implementation.

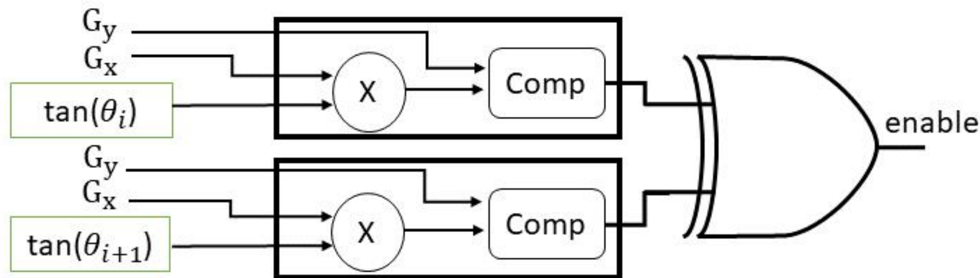

Figure 2.8: A basic hardware architecture for bin assignment. In this architecture, gradient in horizontal direction is multiplied by the tangent limit and is compared with the gradient in vertical direction. The two signals from the comparators (shown as Comp) of two adjacent blocks are XORed. If they are different, an enable signal is issued to write the value into the selected bin.

For creating the histogram, Rettkowski et al. [60] use eight bins for degrees between $-\pi/2$ and $\pi/2$. They also scale the numbers so that instead of division, only integer multiplication and shifting are enough to find the correct bin for each histogram.

Ilas [75] provides slope computation as a replacement for arctangent calculation. Since the angle value is only used for bin assignment, the author suggests that slope value can do the same task with fewer computations. The slope can be computed as a division of gradient in y orientation by the gradient in x orientation. However, since slope changes more rapidly and can have large values, Ilas uses an adapting scaling method to saturate the value of slope in predetermined bins. Therefore, the HOG features computed are completely different than the conventional HOG. But the result shows that it has an accuracy near the original HOG in the case study of car detection systems.

Similar to magnitude computation, some work use IP-cores for orientation computation as well. Meus et

al. [61] and Ngo et al. [64] use CORDIC IP cores for orientation computations.

Observations and Conclusions: The most commonly used method which many papers [68][53][55][69][77][62][63] use is to compute the tangent value of border angles and compare the y-direction gradient with the multiplication of the x-direction gradient and tangent limits. In this way, no trigonometric calculation is required. The method presented by Ilas [75] is similar in that there is no need for trigonometric calculation. However, it has less accuracy than the exact orientation computation in the HOG algorithm.

Bin number selection

Some researchers propose methods to simplify the HOG algorithm. As an example, some papers use less than nine bins for histogram creation. For example, Rettkowski et al. [60] use eight bins, therefore each bin covers a domain of 180/8 degrees. Ilas [76] proposes to use only four bins instead of nine bins in the histogram computing stage, which correspond to 0, -45, +45, and 90 degrees. In this way, only 2.6% of LUTs are used. However, the accuracy is decreased. The author suggests that one can use this algorithm to find candidates and perform the original HOG only on selected candidates to improve the detection time. Chen et al. [80] propose to divide the linear gradients into only six orientation bins in 0° to 180° to reduce the computation complexity, with negligible accuracy loss compared to nine bins. Ilas [74] divides the orientation bins for every 16 degrees instead of 20 degrees. Therefore, this implementation consumes fewer hardware resources, and the division by 20 can be replaced by division of 16 which is implemented conveniently as a 4-bit shifter. As a result, 6% fewer LUTs and 17% fewer registers are used.

Observations and Conclusions: Using a smaller number of bins can reduce hardware utilization; however, it decreases the accuracy in some applications. Therefore, this is suitable for applications where hardware resources are limited but accuracy reduction is tolerable. Furthermore, the method proposed by Ilas [76] which uses a simpler classifier (with four bin HOG) first and a more precise HOG after that, can perform faster by using more hardware resources.

Interpolation

Another area of algorithm simplification is interpolation between bins. Although some authors such as Chen et al. [69] implement bilinear interpolation, Ilas [74] assumes that if no interpolation is considered between bins in both training and testing phases, the accuracy will not change. On the other hand, Chen et al. [69] compute the weight of gradient magnitude for two adjacent bins using (2.10):

$$\alpha = (n + 0.5) - b\frac{\theta(x, y)}{\pi} \tag{2.10}$$

in which, b (the total number of bins) is 9 and n is the bin to which θ belongs. After that, the magnitudes are weighted as (2.11) and (2.12) for two adjacent bins:

$$m_n = (1 - \alpha) \times m(x, y) \tag{2.11}$$

$$m_{nearest} = \alpha \times m(x, y) \tag{2.12}$$

Observations and Conclusions: Performing interpolation in the bin assignment makes histograms smoother. However, Ilas [74] shows that if interpolation is not used in both training and testing phases of feature

extraction, the overall accuracy reduction is negligible. Therefore, interpolation is useful in cases which accuracy is critical, and the hardware utilization and delay caused by this unit are acceptable.

2.3.4 Normalization

The next step in the HOG algorithm is the normalization of histograms in each block. After assigning the magnitudes to different bins, the histograms of cells inside a block are normalized. Several papers propose techniques to make this normalization computation efficient.

Mizuno et al. [14], Chen et al. [69], Ma et al. [52], and B.K. et al. [82] use the Newton method to approximate L2 normalization. Chen et al. [69] and B.K. et al. [82] use IEEE754 standard floating-point representation to normalize the histograms in each block. The authors also use the Newton-Raphson method to approximate inverse square root.

The formula for Newton approximation is shown in (13):

$$\frac{1}{\sqrt{x}} = y_d \times \frac{(3 - x(y_d)^2)}{2} \tag{2.13}$$

where

$$y_d = \text{Decimal}\{(x_{IEEE754} - 0x5F3759DF\} \tag{2.14}$$

where $x_{IEEE754}$ is the IEEE754 floating-point representation of x. Decimal{h} is the decimal representation of the hexadecimal value of h. Also, $0x5F3759DF$ is the magic number [67] for Newton-Raphson approximation so that there is no need for iteration.

Wang et al. [62] use shifting instead of division for block normalization according (2.15).

$$h_n = h >> [log_2(sum(h))] \text{ for } h \neq 0 \tag{2.15}$$

where $sum(h)$ is the summation of values in vector h and h_n is the normalized vector.

Observations and Conclusions: Although the method provided by Wang et al. [62] consumes less hardware resources since it uses only shifting and addition, it is less accurate. The Newton approximation method which is used in some work [14][69][52][58][82] for inverse square root is more often used for normalization since it consumes fewer hardware resources than computing the more exact value of inverse square root using IP cores.

2.4 Data manipulation techniques

In this section, we review the methods which change the parameters that affect the whole algorithm. In section 2.4.1, bit-width and numerical representation and their effects on FPGA implementation are discussed. Then, in section 2.4.2, the methods that contribute to data path optimization of the HOG algorithm are surveyed. Finally, in section 2.4.3, the methods which are more focused on memory usage optimization are presented.

2.4.1 Numerical representation

One of the main techniques for efficient implementation of mathematical algorithms on an FPGA is choosing an appropriate numerical representation. If more than the necessary number of bits are used in the implementation, without any changes in accuracy, more memory is used, and the total latency of the circuit is increased. On the other hand, if the bit-width is too small or the representation is too simple, it will result in accuracy loss.

While some researchers such as Komorkiewicz et al. [49], Chen et al. [69], and B.K. et al. [82] use floating-point representation similar to that of the original HOG algorithm [15], others such as Ma et al. [54], Hemmati et al. [55], and Ngo et al. [64] use a constant number of bits for fixed-point representation for HOG. Ma et al. [54] investigate using fixed-point calculations for HOG feature extraction. The authors suggest that using 13-bit fixed-point can preserve the accuracy of an HOG-SVM pedestrian detector and even enhance it. Although reducing bit-width decreases area consumption, it may not preserve accuracy for some applications.

Some research groups use different fixed-point representations for different parts of the HOG algorithm [48][14][52]. Negi et al. [48] use 19 bits for gradient calculations, 14 bits for each histogram, and 33 bits for normalized histograms. Although they achieve 62.5 frames per second, due to the fixed-point implementation, the accuracy of their model decreases. Mizuno et al. [14] dedicate 9 bits for gradient magnitude, 6 bits for gradient orientation, 11 bits for orientation histogram, and 25 bits for L2 normalization which they use Newton method to approximate it. In order to preserve the accuracy, first, Chen et al. [52] use 27-bit fixed points for HOG calculations. Then, for each part of the calculations, they decrease the bit-width and compare the results with software implementation to determine if the accuracy has decreased or not.

Observations and Conclusions: Overall, data representation is a trade-off between accuracy, speed, power, and resource consumption. The methods which use floating-point numbers [49][69][82] obviously lead to higher accuracy. However, floating-point computation requires more hardware resources and is not as fast as fixed-point computations. Therefore, although the ones that use fixed-point calculations [54][55][64] are less accurate, they perform faster and use fewer FPGA resources than the others. On the other hand, methods such as those proposed by Negi et al. [48], Mizuno et al. [14], and Ma et al. [52] are more balanced since they try to optimize the hardware usage and also preserve the accuracy of the model.

2.4.2 Data flow modification

In this section, we review the methods which optimize the data path of the HOG algorithm. Mizuno et al. [14] present an architectural study on HOG feature extraction. The authors propose a cell-based pipeline for HOG computations which reduces memory bandwidth. They use an external CPU to control the pipeline for HOG computation. They process images with 1920x1680 pixels and achieve 30 frames per second with 76 MHz clock frequency. Komorkiewicz et al. [49] propose using a 32-bit single-precision floating-point numbers in HOG computation for object detection. They use a complete pipeline without the need for memory in intermediate calculations and achieve 60 frames per second for 640x480 frames. Their intent is to improve accuracy by using more hardware resources.

Luo et al. [63] implement the HOG algorithm fully on an FPGA. They test their design on 800x600 images with 150¬ MHz clock frequency. Three line-buffers of 800 8-bit words are used to accommodate the pixels required for calculating the gradients simultaneously. After assigning magnitudes to the bins, the cell

values are computed using several shift registers on the fly, and the result is given to a block normalization module. The authors use FIFO memories to save the blocks which overlap with other blocks, and use them later without redundant computations.

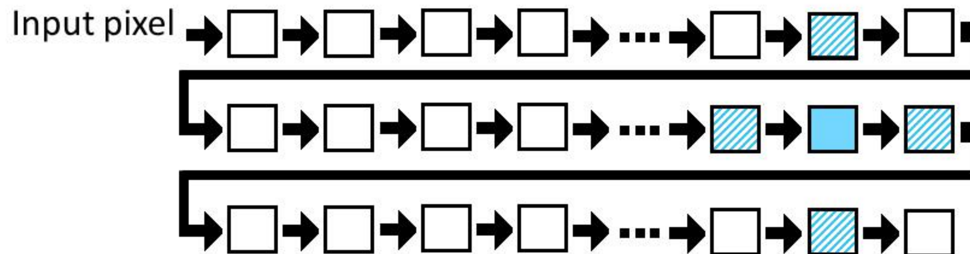

Figure 2.9: A pipeline of three rows of line-buffer registers [80][63].] The pixels enter the registers from the top left register and flow into the buffer. At each clock cycle, the solid blue register is the main pixel and the values of dashed pixels on top, bottom, left, and right of the main pixel, which have fixed positions, are used to compute the gradients.

Chen et al. [80] propose to use a shift register which covers three rows of pixels since HOG feature extraction for each pixel requires four other surrounding pixels. By using buffers for every three lines, after an initial setup time and when the buffers are full, at each clock cycle, the required pixels are available. Therefore, at each clock cycle, the values of the required pixels are extracted from the shift registers, and the computations are done. Fig. 2.9 shows the architecture of line buffers in that paper. Finally, when the orientation of each pixel is computed in the HOG engine, the L1 distance of the HOG features and the HOG of the object model are computed to classify the object. The line buffers enable the algorithm to run faster but on-chip registers are used for those buffers. Using a fewer number of bins allows the algorithm to run faster however it also decreases its accuracy.

Qasaimeh et al. [65] propose a systolic array structure for HOG computation. For sliding window operation, they propose to compute the histograms for each 3x3 window in an overlapping manner. When the histogram of one window is computed, for the next window, the contribution of the last column is subtracted from the histogram, and the contribution of the new column is added. Therefore, at each window, one column calculation is preserved. Fig. 2.10 shows how this method affects the computed histogram.

Observations and Conclusions: Data path optimization techniques can affect the overall speed of the implementation. Designing the circuit using pipeline architecture can reduce the clock period and increase maximum clock frequency. On the other hand, using line buffers is a common solution for parallelizing the input data which is read at the beginning of the algorithm. These techniques lead to performance enhancement in speed.

2.4.3 Memory optimization

Hemmati et al. [55], Ma et al. [52], Chen et al. [80], and Luo et al. [63] propose innovative methods for memory usage. One of the innovations of the work provided by Hemmati et al. [55] is the use of four different memories, each containing one cell of each block. Therefore, gradient calculation, histogram generation, and block normalization can be done in parallel by accessing the cell values from four independent memories.

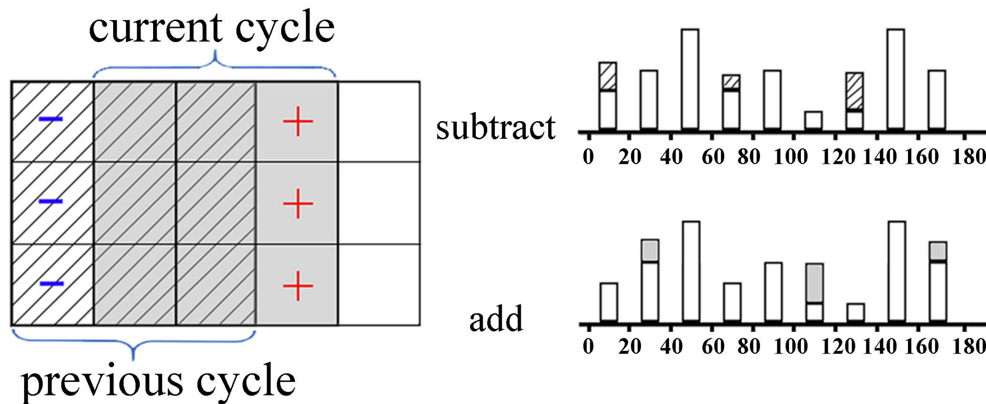

Figure 2.10: Sliding window proposed by Qasaimeh et al. [65]. The left image shows two consecutive cycles. In the current cycle, the contribution of pixels from left side of the previous window (blue -) are subtracted from the histogram and the contribution of new pixels (red +) are added. The top right image shows that some values are subtracted from bin values and in the bottom right image new values are added. The number under each bin limit shows the ranges of the orientations of that bin.

Ma et al. [52] design the HOG hardware to fetch pixels from two rows of cells and to process two rows at the same time to have more parallelism. Since cells in one row can be used in the next row for block normalization, alternating between odd and even rows can prevent computing the histograms two times and therefore lead to a speed-up in their implementation. In addition, the authors pack each pair of magnitude and orientation into a single 32-bit integer. In each memory access, a 64-bit value is returned which is two pairs of magnitude and orientation. Still, in their implementation, they compute the magnitude and orientation in software and send them to the FPGA for further computations and block normalization. They attempt to maintain the accuracy of the HOG algorithm by not reducing the number of bits, the number of bins, and scaling factor for classification. They propose a method to make the whole algorithm (without simplification) faster. However, they use software for more complex computational parts such as magnitude and orientation computation. The main goal of their work is to preserve the accuracy and then improve the speed of their design.

Observations and Conclusions: Memory usage optimization has an important role in data throughput. However, storing the data as required in the memory might not be achievable in all situations, since it is dependent on the application and platform.

2.5 Modified HOG-based features

In order to increase the accuracy of the overall algorithm, some papers have modified the HOG features. In this section, we review four methods which implement modified HOG algorithm on FPGAs.

2.5.1 Feature compression

Advani et al. [50] propose feature compression. From the 72 features that are produced after block normalization in their paper, a concatenation of 18 features (accumulated bins for each cell), 9 features (coupling 18 bins into nine bins) and 4 features (accumulating bins of each cell) are generated. Overall every 72 features

are compressed into 31 features. Therefore, the final classifiers (SVM in this case) only have to process 31 features.

Rettkowski et al. [60] propose to transform the final HOG features to binary values. They convert the features to binary values by comparing each value of the histograms with 8/128 (which is a 4-bit shift) and transform every 14-bit value in the histograms into a 1-bit feature, which is very beneficial in memory usage reduction.

2.5.2 Correlation of HOG features

Nishizumi et al. [58] introduce sparse FIND (Feature Interaction Descriptor) features which are extracted from HOG features. They first compute the HOG features and then calculate FIND features by obtaining the correlation between HOG features and normalizing the correlations. Because of the complexity of the calculations, they only extract the elements with high validity in identification from HOG (which have values more than a threshold). They calculate this threshold using (2.16) where k is a sparsification parameter, m is the number of histogram bins in the block, and hi is the HOG value in each bin. Equation (2.17) also shows the parameter which is used for normalization.

$$\text{threshold} = (\frac{k}{m}) \sum h_i \tag{2.16}$$

$$\alpha = \frac{1}{\sum {h_i}^2} \tag{2.17}$$

Finally, they compute the correlation as shown in (18).

$$f(h_i, h_j) = \alpha \times h_i \times h_j \quad \text{ if } h_i, h_j > \text{threshold} \tag{2.18}$$

where h_i and h_j are values of the bins in the histogram that are bigger than the threshold. They use the correlation to reduce the number of dimensions. Using these correlation values as new features reduces the number of previous features, and the classification step becomes simpler. However, for computing sparse FIND features, multiplication and division are required which consume more hardware resources. Their method improves the accuracy but consumes more area in the feature extraction part.

2.5.3 Histogram of significant gradients

Bilal et al. [59] introduce HSG (Histogram of Significant Gradients) which is a modified version of HOG. In this method, they compute the average magnitude in each cell, and if a gradient magnitude is higher than the average value, that magnitude casts a binary vote to the histogram bin (the value of that bin is incremented by one unit). One advantage of this method is that there is no need to normalize HOG features and another advantage is that the final feature vector contains integer values only, hence, the hardware computation is simplified.

2.5.4 Gradient extension

Sledevič et al. [51] increase the number of HOG features by using pixels which are one pixel further from the center pixel. Their implementation uses both the 8 pixels around the main pixel and the 16 pixels around

the first 8 pixels to compute the gradient, and this method processes 24 pixels for each pixel. Then for each block, they add the value to the bin dedicated to the specific direction. Fig. 2.11 shows the histogram bins for one pixel using the method provided by Sledeviè et al. [51]. Higher accuracy is expected by using more features to get more information about the local variations of the image. They use the HOG algorithm for tracking objects.

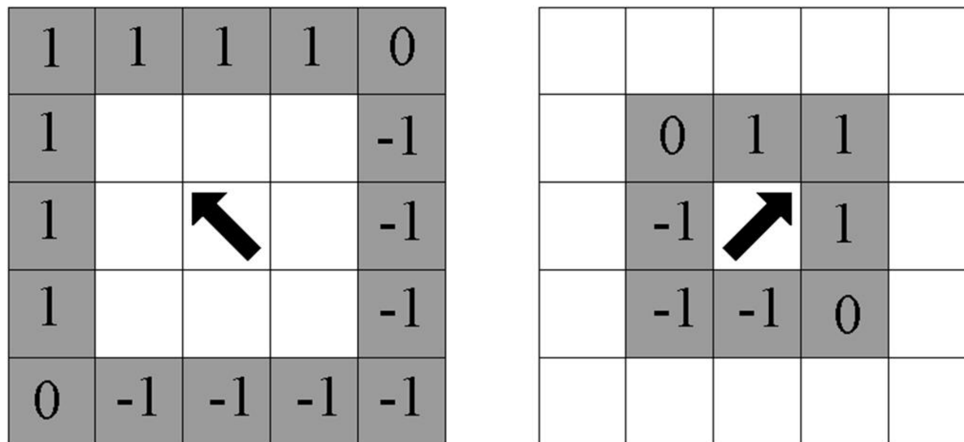

Figure 2.11: The extension of HOG features used by Sledeviè et al. [51]. The left image shows the example orientation in the main pixel calculated by 16 pixels with 1-pixel distance from the main pixel. The right image shows the example orientation of the pixels around the main pixel. The histograms derived by both methods are concatenated together as final features.

Observations and Conclusions: In this section, we reviewed four variants of the HOG algorithm and their hardware implementation. The methods described in the feature compression subsection can be used where FPGA resources are limited or the features are to be sent through a communication channel (such as IoT devices). Sparse FIND features require more computations for feature extraction. Although the authors decrease the number of features, more hardware recourses are utilized to compute them. HSG features simplify the histogram computation calculations by creating binary features which is a trade-off for accuracy. In contrast, extension of gradients for 16 further pixels uses more hardware to have a better accuracy for their application.

2.6 Hardware-software implementations

While some work focus on pure FPGA implementation of the HOG algorithm, others propose a hardware-software co-design approach. Mizuno et al. [14] use CPU to control the logic of a pipeline for HOG computations. For intermediate computations such as storing histograms of processed cells and the SVM coefficients, they use an external SRAM memory. Ma et al. [52] implement the HOG-SVM system on a Convey HC-2ex platform, which is composed of two Intel Xeon four-core processors and four Xilinx Virtex-6 FPGAs that can communicate with each other through shared memory. They compute the magnitude and orientation in software and store them in memory. Then, the FPGA cores read the data from the memory for further computations and block normalization.

Rettkowski et al. [60] implement a software version, a HW/SW codesign version, and a hardware version of the HOG algorithm. In their paper, they show that hardware design is 503x faster than the software version and the HW/SW version is 9x faster than the software version. For the HW/SW version, they use the Xilinx tool SDSoc to convert C code to hardware implementation. They compute the gradient, magnitude and orientation in software, and histogram generation and block normalization in hardware. The main contribution by Huang et al. [71] is in the classification part which they classify all blocks with the Adaboost classifier, and give only the most probable candidate windows (which have 36x(number of blocks) features) to a linear SVM for human detection applications. The authors implement HOG as a hardware accelerator to be used in a HW/SW system. They send one block of pixels at each time to the FPGA to get the normalized histograms of that block. Then, they perform the classification using an ARM processor. Ngo et al. [64] implement the HOG algorithm including the sliding window step in hardware and the classification step in software. Most of the work in our survey use Hardware Description Languages such as Verilog or VHDL. However, B.K. et al. [82] and Meus et al. [61] use high level synthesizers for FPGA implementation.

Observations and Conclusions: Deciding whether or not to use HW/SW implementation depends on the application and the other parts of the design. As mentioned by Rettkowski et al. [60], pure hardware implementation is usually faster than HW/SW version. However, if programmable logic is limited in a circuit, software solutions can be useful. In applications and scenarios where FPGA resource usage is more critical than speed, the HW/SW design approach performs better. However, it is important to separate the algorithm into the appropriate parts for each platform to get the best results.

2.7 Discussion

In this section, first, we compare the performance of different work with respect to frame rate and speed. Then, in the second part, we suggest design guidelines for optimized implementation of the algorithm while considering the limitation of the system.

2.7.1 Speed comparison

In Table 2.4 and Table 2.5, we summarize the results for the surveyed papers for HOG implementation. One of the most commonly used metrics for speed evaluation of an image processing algorithm is the number of frames processed in one second. However, since FPGA implementation of an algorithm depends on the clock frequency and the size of the input image, frames per second may not be a fair evaluation metric for comparing different algorithms. The reason for this argument is that in the same design, frames per second can easily be increased by decreasing the size of the input frame or by increasing the clock frequency. Therefore, we use the method presented by Ngo et al. [64] which computes the pixels per clock cycle, according to (2.19) and (2.20).

$$\text{pixels per clock cycle} = W \times H \times FR \times \frac{1}{f} \tag{2.19}$$

$$\frac{\text{pixels}}{\text{clock cycle}} = \frac{\#\text{ of pixels}}{\text{frame}} \times \frac{\text{frame}}{\text{s}} \times \frac{\text{s}}{\text{clock cycles}} \tag{2.20}$$

Table 2.4: Speed comparison and FPGA used in each reference

Reference	FPGA Platform	Frame rate (fps)	Pixels per clock cycle
Long et al. [78]	Altera Stratix IV	10000	8.192
Nishizumi et al. [58]	Virtex Ultrascale	46	1.063
B.K. et al. [60]	Zynq	39	1.001
Zhou et al. [77]	Zynq	236000	0.999
Ngo et al. [64]	Cyclone V	526	0.997
Hemmati et al. [55]	Zynq	60	0.995
Zhou et al. [77]	Zynq	116	0.995
Meus et al. [61]	Zynq	60	0.747
Komorkiewica et al. [49]	Virtex-6	60	0.737
Sledevie et al. [51]	Virtex-4	60	0.737
Advani et al. [50]	Virtex-7	30	0.622
Luo et al. [63]	Cyclone IV	162	0.518
Hahnle et al. [53]	Virtex-5	64	0.498
Mizuno et al. [14]	Cyclone IV	72	0.454
Bilal et al. [59]	Cyclone IV	162	0.422
Chen et al. [80]	Zynq	41	0.251
Xu et al. [56]	Spartan-6	47	0.225
Wang et al. [62]	Cyclone IV	60	0.170
Ma et al. [52]	Virtex-6	68	0.139
Negi et al. [48]	Virtex-5	62	0.108
Blair et al. [68]	Virtex-6	13	0.051
Ahmad et al. [84]	Virtex-6	6	0.002
Hsiao et al. [70]	Spartan-6	15	0.001

where W is the width of the frame, H is the height of the frame, FR is the frame rate, and f is the frequency. In order to compute this metric, first, the frames per second rate is multiplied by the input image size and the result is the number of pixels processed in each second. Then, the obtained value is divided by the clock frequency of the system so that all work can be compared in the same clock domain. Table 2.4 shows this measure for different implementations and hardware platforms. We compute this metric only for papers that reported the frame size, the clock frequency of the FPGA, and frames per second rate.

Pixels processed per clock cycle is shown in this Table. The FPGA platform is also shown for comparison.

As shown in Table 2.4, some work use Zynq family FPGAs [55][77][80][60][61][81] while others use Virtex series FPGAs [48][49][68][53][52][50][58][51][84]. On the other hand, [14][59][62][63][64] use Cyclone family FPGAs. Cyclone V devices have more available memory while

Virtex family FPGAs have more logic elements than Cyclone and Zynq series. It can be seen that methods which are implemented on Zynq series are mostly in the upper half of Table 2.4 indicating their speed superiority. In addition, the results by Ngo et al. [64] who use Cyclone V FPGA is better than all Cyclone IV FPGAs. Although it seems that newer FPGAs and technology lead to faster systems, it cannot be concluded as a fact since the works with newer technology are the most recent ones and they have more effective innovations in their implementation as well.

Nishizumi et al. [58] use the CORDIC method for arctangent and square root operations for histogram generation. They also use the Newton method for square root division for normalization. While most of the work read the input data as one pixel per clock cycle, Long et al. [78] use a high-speed camera and receive the data by 64 pixels per clock cycle. Therefore, the overall speed of their system is higher than the others.

The work proposed by Ngo et al. [64] has a frame rate of 526 which is higher than other projects with similar input stream size and throughput. The reason is that they use buffers for data path optimization and CORDIC IP cores for magnitude and orientation computation. As a trade-off, they use more registers than others. The next best performance with respect to frame rate are found in [63], [59], and [77]. Zhou et al. [77] use a pipeline architecture for HOG implementation and simplification on normalization formula. They also report the frame rate based on 32x32 frames which are much smaller than what others use and their frequency is 241 MHz which is higher than other work. Luo et al. [63] use Altera IP cores for square root and line buffers for data path structure. Bilal et al. [59] remove the normalization step by using a threshold value in the bin assignment stage. In conclusion, data path optimization techniques, such as using pipeline architecture and line buffers, have a great effect on the final frame rate. Besides, the normalization step is one of the most time-consuming steps of HOG algorithm implementation.

2.7.2 Resource utilization

The resource utilization of different approaches for FPGA-based implementation of HOG is shown in Table 2.5. These resources, which are normally used to compare designs, include look-up tables (LUT), block RAMs (BRAM) and digital signal processing units (DSP) on the FPGA. LUTs are the smallest programmable elements on the FPGA. Block RAMs are memories which are embedded on the FPGA. DSPs on the FPGA are used for mathematical operations such as multiplications. Since many papers do not report their hardware utilization explicitly, we include only the ones which report their results in Table 2.5. The work proposed by Rettkowski et al. [60] is on top of the table. This work does not use block memories but uses more LUTs instead. The method proposed by Bilal et al. [59] does not require the normalization step which has led to less resource usage. Considering the work which use most hardware resources, the method proposed by Ma et al. [52] has the most LUT and BRAM usage in Table 2.5 since they implement the HOG algorithm for 34 scales. Advani et al. [50] implement multi-scale detection while Komorkiewicz et al. [49] implement

This table is sorted in an increasing order of BRAM (Block RAMs) usage. The second column shows that the hardware resources reported are for a full system or only the HOG core. Full systems usually contain a classifier such as SVM or Adaboost. The ones with SoC have used a processor besides the HOG core.

HOG using floating-point numbers. Li et al. [79] implement a highly parallel system which receives the input of 64 pixels per clock cycle. They also implement multi-scale detection. These are the main reasons for the large number of used resources in these papers.

2.7.3 Design guidelines

The most important criteria in embedded real-time systems are speed, accuracy, resource usage, and power. In this part, we summarize the techniques reviewed in this survey as a design guideline based on these criteria. Since FPGA-based designs are typically of lower power than CPUs and GPUs, in this section we focus on speed, accuracy and resource utilization of the designs. Table 2.6 shows the methods suggested for increasing the speed of the final design by using different techniques in existing work. Methods described in this table require a fewer number of computations and use more hardware resources. However, they are optimized for speed. The methods described in Table 2.7 are optimized for accuracy. No simplification is used for the methods in Table 2.7 in order to have the same features as the the software implementation. The methods described in Table 2.8 are optimized for the cases which hardware resources are limited.

Table 2.5: Resource utilization

Reference	Implementation	FPGA	LUT	BRAM (Kbit)	DSP
Rettkowski et al. [60]	Only HOG core	Zynq	18987	0	4
Bilal et al. [59]	Only HOG core	Cyclone IV	751	43.7	0
Long et al. [78]	Full system	Stratix IV	266023	47	236
Bilal et al. [59]	Full system	Cyclone IV	65501	103	10
Xu et al. [56]	Only HOG core	Spartan-6	9955	208	66
Ngo et al. [64]	Only HOG core	Cyclone V	8610	326	74
Mizuno et al. [14]	Only HOG core	Cyclone IV	34403	334	68
Yu et al. [57]	Full system (SoC)	Spartan-6	15167	351	19
Hemmati et al. [55]	Only HOG core	Zynq	7649	432	26
Ngo et al. [64]	Full system (SoC)	Cyclone V	12138	437	65
Hahnle et al. [53]	Only HOG core	Virtex-5	5188	1188	49
Negi et al. [48]	Full system	Virtex-5	17383	1327	N/A
Chen et al. [80]	Full system	Zynq	10052	1620	2
Ranawaka et al. [73]	Only HOG core	Zynq	6069	2682	117
Adiono et al. [72]	Only HOG core	Cyclone IV	83000	2800	90
Wang et al. [62]	Only HOG core	Cyclone IV	94374	3042	N/A
Blair et al. [68]	Full system	Virtex-6	108518	3744	138
Blair et al. [81]	Full system	Virtex-6	77623	3906	108
Komorkiewicz et al. [49]	Full system	Virtex-6	113359	4284	72
Raj et al. [85]	Full system	Virtex-5	69120	4680	N/A
Li et al. [79]	Full system	Stratix IV	313349	9696	268
Advani et al. [50]	Full system	Virtex-7	137361	13500	202
Ma et al. [52]	Full system	Virtex-6	184953	13737	190

Table 2.6: Optimization for speed

Parameters	Original Form	Technique	Reference
Magnitude	$Mag(x, y) = \sqrt{{G_x}^2(x, y) + {G_y}^2(x, y)}$	$Mag(x, y) = \lvert G_x(x, y)\rvert + \lvert G_y(x, y)\rvert$	[80] [80]
Orientation	$\theta_i \leq arctan(\frac{G_y(x,y)}{G_x(x,y)}) < \theta_{i+1}$	$G_x(x, y)tan(\theta_i) \leq G_y(x, y) < G_x(x, y)tan(\theta_{i+1})$	[68]
Normalization	$h_n = \frac{h}{\sqrt{\lvert\lvert h\rvert\rvert^2+e}}$	$h_n = h >> [log_2(sum(h))]$ for $h \neq 0$ [62]	
Bit-width	Floating-point number	13-bit fixed-point	[54]
Data path	—	Four different memories each for one cell in a block	[55]
	—	Line buffers for lines of an image	[52][80][63]
	—	Pipeline stages of HOG	[14][49]
Simplifications	—	Single-scale	[48]
	—	No interpolation	[74]

* The symbols in the table are: h_n = normalized histogram of a block, h = histogram of a block before normalization, G_x = gradient in horizontal direction, G_y = gradient in vertical direction, θ_i = angle limits (normally from 0 to 180 and in 9 steps).

2.8 Conclusion

In this paper, we reviewed the methods used to implement the histogram of oriented gradients algorithm on FPGAs in the past decade (2010-2019). Some of the reported techniques are related to individual steps of

Table 2.7: Optimization for accuracy

Parameters	Original Form	Technique	Reference
Magnitude	$Mag(x,y) = \sqrt{{G_x}^2(x,y) + {G_y}^2(x,y)}$	$Mag(x,y) = \sqrt{{G_x}^2(x,y) + {G_y}^2(x,y)}$	[15]
Orientation	$\theta_i \leq arctan(\frac{G_y(x,y)}{G_x(x,y)}) < \theta_{i+1}$	$G_x(x,y)tan(\theta_i) \leq G_y(x,y) < G_x(x,y)tan(\theta_{i+1})$	[68]
Normalization	$h_n = \frac{h}{\sqrt{\|\|h\|\|^2+e}}$	$h_n = \frac{h}{\sqrt{\|\|h\|\|^2+e}}$	[15]
Bit-width	Floating-point number	Floating-point number	[49][69][82]
Simplifications	—	Multi-Scale	[68][52][79]
	—	Use interpolation in bin assignment	[69]

* The symbols in the table are: h_n = normalized histogram of a block, h = histogram of a block before normalization, G_x = gradient in horizontal direction, G_y = gradient in vertical direction, θ_i = angle limits (normally from 0 to 180 and in 9 steps).

Table 2.8: Optimization for hardware resource utilization

Parameters	Original Form	Technique	Reference
Magnitude	$Mag(x,y) = \sqrt{{G_x}^2(x,y) + {G_y}^2(x,y)}$	$Mag(x,y) = \|G_x(x,y)\| + \|G_y(x,y)\|$	[80]
Orientation	$\theta_i \leq arctan(\frac{G_y(x,y)}{G_x(x,y)}) < \theta_{i+1}$	$\theta_i \leq arctan(\frac{G_y(x,y)}{G_x(x,y)}) < \theta_{i+1}$	[68]
Normalization	$h_n = \frac{h}{\sqrt{\|\|h\|\|^2+e}}$	$h_n = h >> [log_2(sum(h))]$	[62]
Bit-width	Floating-point number	13-bit fixed-point	[54]
Simplifications	—	Single-scale	[48]
	—	No interpolation	[74]
	—	A smaller number of bins	[80]
	—	Binary image input	[76]

* The symbols in the table are: h_n = normalized histogram of a block, h = histogram of a block before normalization, G_x = gradient in horizontal direction, G_y = gradient in vertical direction, θ_i = angle limits (normally from 0 to 180 and in 9 steps).

the algorithm, and some affect the whole algorithm. We also reviewed different simplification methods and the hardware implementation of modified features which were based on the original HOG algorithm. After that, we compared the recent work regarding speed, accuracy, and resource utilization of the designs. It was observed that the methods which focus on optimizing the data path of the design have the most effect on the speed of the circuit. Finally, we presented three design guidelines for FPGA-based HOG implementation which categorize different methods regarding the limitation and requirements of different applications.

The research community has considered many aspects of the hardware implementation of the HOG algorithm. However, many have not reported the accuracy of their proposed methods, possibly due to the fact that their main focus lies in other aspects such as processing speed and resource utilization, or accuracy is not a concern in their particular application. For the ones that reported accuracy, it is often less than that of a software implementation. Interpolation in bin assignment and normalization steps are the parts that most of the work approximated and simplified to gain higher speed, hence, leading to accuracy issues. Though, these two steps have great potential to enhance FPGA-based HOG implementation in a future

work. Since the proposed methods for multi-scale detection are not the focus with respect to hardware resource consumption, one of our future directions is to find an optimum number of image scales to achieve higher accuracy while maintaining other design metrics.

Chapter 3

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

The histogram of oriented gradients is a commonly used feature extraction algorithm in many applications. Hardware acceleration can boost the speed of this algorithm due to its large number of computations. We propose a hardware–software co-design of the histogram of oriented gradients and the subsequent support vector machine classifier, which can be used to process data from digital image sensors. Our main focus is to minimize the resource usage of the algorithm while maintaining its accuracy and speed. This design and implementation make four contributions. First, we allocate the computationally expensive steps of the algorithm, including gradient calculation, magnitude computation, bin assignment, normalization and classification, to hardware, and the less complex windowing step to software. Second, we introduce a logarithm-based bin assignment. Third, we use parallel computation and a time-sharing protocol to create a histogram in order to achieve the processing of one pixel per clock cycle after the initialization (setup time) of the pipeline, and produce valid results at each clock cycle afterwards. Finally, we use a simplified block normalization logic to reduce hardware resource usage while maintaining accuracy. Our design attains a frame rate of 115 frames per second on a Xilinx® Kintex® Ultrascale™ FPGA while using less hardware resources, and only losing accuracy marginally, in comparison with other existing work.

3.1 Introduction

Feature extraction is one of the main stages in a wide variety of pattern recognition applications [19], [86], [87] . In particular, feature extraction and description has been used in numerous computer vision algorithms for many applications. There have been many feature description algorithms proposed in the past decade, such as SIFT (scale-invariant feature transform) [19] and SURF (speeded up robust features) [16], which have all shown outstanding results in a wide variety of applications. HOG (histogram of oriented gradients) [15] is one of the commonly used descriptors which has proven to be useful in many computer vision applications, including human detection, car detection, and general object recognition.

An object detection system is typically a combination of an input sensor, a feature extraction module and a classifier to make decisions based on the extracted features. Since the main application of the HOG features is in human and object detection, the output of a camera sensor is given to the HOG descriptors, usually followed by a suitable classifier such as SVM (support vector machines). The main drawback of the HOG algorithm is its computational complexity, which prevents it from meeting the timing requirements of some practical applications. Therefore, many researchers have tried to implement this algorithm on hardware platforms such as GPUs (graphical processing units) and FPGAs (field programmable gate arrays) to reap the benefits from parallel computation and thus improve speed.

Ma et al. [52] compared the CPU (central processing unit), GPU, and FPGA implementation of the HOG algorithm. They implement the HOG algorithm on an Intel® Xeon® E5520 CPU processor, an Nvidia® Tesla® K20 GPU, and a Xilinx® Virtex®-6 FPGA. Their FPGA implementation consumes 130× less energy than the CPU and 31x less energy than the GPU to process a single frame, while the speed is about 68x better than the CPU and 5x better than the GPU.

Since FPGA implementations typically consume less power than GPUs and CPUs, there has been considerable interest in FPGA implementation in numerous applications [88]–[90]. In particular, many scholars have contributed to FPGA implementation of the HOG algorithm [63], [65], [78], [81], [91]. Hardware implementations are usually evaluated by the four main metrics of speed, accuracy, power consumption and resource utilization. Since there are trade-offs between these metrics, many researchers aim to optimize a single metric depending on the specific application. One way to optimize these metrics is to benefit from the advantageous features of both hardware and software in implementing the algorithm. In these methods, the algorithm is partitioned into different functional stages and the most computationally complex stages are implemented on the FPGA. The stages that are sequential in nature or are controlling the data flow can be allocated to the CPU.

In this work, we propose a hardware–software co-design of the HOG algorithm. Our implementation consists of a fully pipelined HOG-SVM IP-core which is controlled by a MicroBlaze™ processor. MicroBlaze™ is a soft microprocessor core designed for Xilinx® FPGAs. This design benefits from both the computational efficiency of the hardware and the simplicity of control mechanisms in software. Our method preserves accuracy and speed while decreasing resource utilization. Figure 3.1 shows the KCU105 FPGA board and the test environment of this research project. A sample image from the INRIA dataset [92] and the block diagram of the whole proposed system in Vivado® software which we used are shown on the display. We used the UART (universal asynchronous receiver/transmitter) port to load the image in the system as a matter of experimental convenience. We could input the image in any other way and the results would be the same.

We have made four contributions in this paper. The first is an efficient task allocation between hardware and software. The second is a logarithm-based bin assignment in the HOG algorithm. The third is a hardware design for computing the histograms using two parallel modules. Finally, the fourth contribution is the approximation of the normalization level which preserves the accuracy of the system while reducing the hardware resource consumption.

In the rest of this paper, we introduce the HOG algorithm briefly in section 3.2. Then, we review existing works that focus on hardware–software co-design of the HOG algorithm in section 3.3. Then, we introduce our hardware–software co-design method in section 3.4. In section 3.5, we provide the details of our implementation. In section 3.6, we compare the results of our design with other work and discuss the

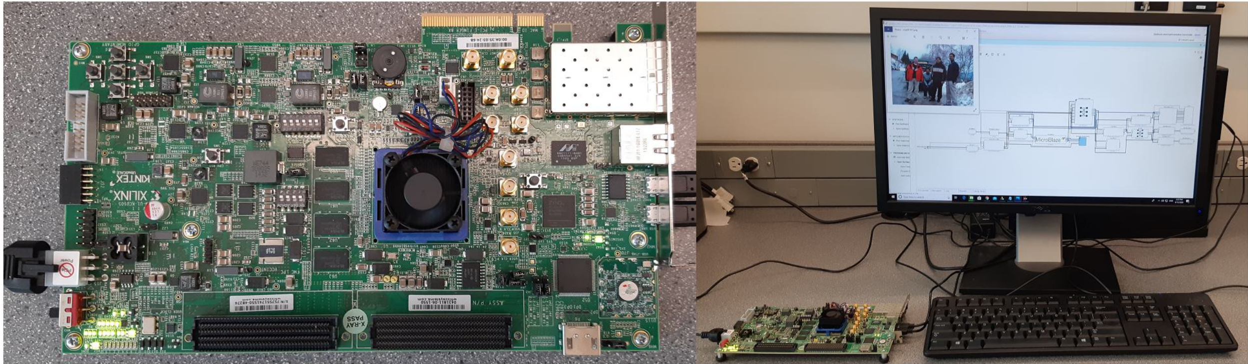

Figure 3.1: The KCU105 FPGA board (left) connected to the computer station (right) for experiments.

advantages and disadvantages of our design. Finally, in section 3.7, we provide conclusions and future work directions.

3.2 Review of the algorithm

In this section, we review the HOG algorithm and the SVM classifier briefly in sections 3.2.1 and 3.2.2, respectively.

3.2.1 The HOG algorithm

The HOG algorithm has several steps, as shown in Figure 3.2.

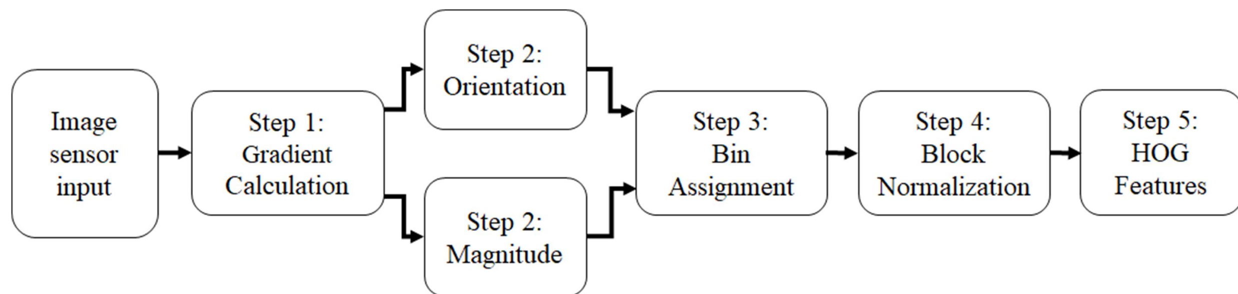

Figure 3.2: A flowchart of the HOG algorithm from input image sensor to HOG features.

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 Equations (3.1) and (3.2):

$$G_x(x, y) = I(x + 1, y) - I(x - 1, y) \tag{3.1}$$

$$G_y(x, y) = I(x, y + 1) - I(x, y - 1) \tag{3.2}$$

where I(x,y) represents the image pixel located in x and y coordinates, and G_x and G_y indicate the gradients of the horizontal and vertical directions, respectively.

In the second step, the magnitude of the gradients is computed as shown in Equations (3.3). 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 Equations (3.4).

$$Magnitude(x, y) = \sqrt{{G_x}^2 + {G_y}^2} \tag{3.3}$$

$$Orientation(x, y) = tan^{-1}(\frac{G_y}{G_x}) \tag{3.4}$$

As shown in Figure 3.2, the next step adds the magnitude values to the bins according to the orientation of each pixel, for histogram generation. In the fourth step, 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 Equations (3.5):

$$h_n = \frac{h}{\sqrt{\sum |h_i|^2 + \epsilon}} \tag{3.5}$$

where h_n is the normalized histogram, h_i is the value of each bin, h is the initial histogram, and ϵ is a very 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 16 pixels (4×4) form a cell and every four cells (2×2) form a block. Figure 3.3 represents this hierarchy. In Figure 3.3, the orientation of the gradient for each pixel is shown by arrows. The boldness and size of the arrows represent the magnitude of the gradients for that pixel.

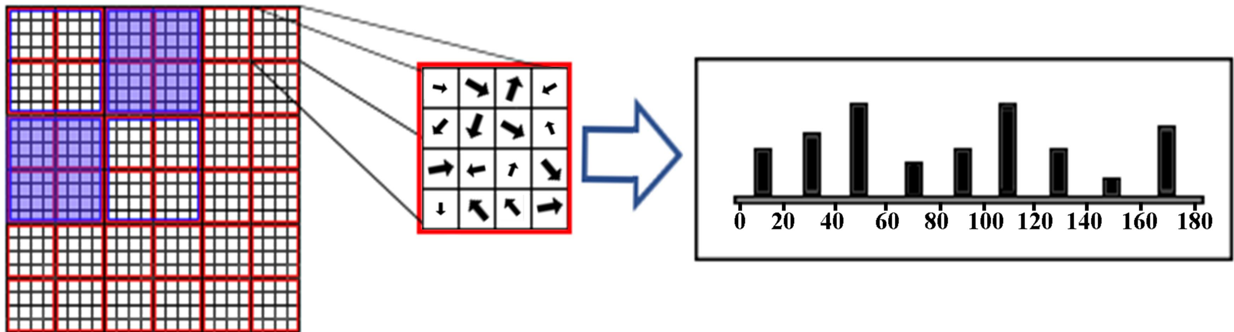

Figure 3.3: Visualization of cells (4 by 4 pixels) and blocks (each containing 4 cells) in the HOG algorithm.

3.2.2 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 [15]. 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, which is used in many applications. In the 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 (3.6):

$$f(x) = \sum {w_i}^T x + b \tag{3.6}$$

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 testing phase in an SVM classifier, it is a popular choice for hardware implementation.

3.3 Related work on hardware–software 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 previous work [37], [38]. 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.

3.3.1 Hardware implementation of the HOG algorithm

There are several implementations of the HOG algorithm using pure hardware [63], [65], [78], [91]. One of the benefits of implementing the HOG algorithm on hardware is of course speed enhancement. Implementing the whole algorithm on hardware is beneficial when resource consumption is not a constraint.

Qasaimeh et al. [65] propose a systolic architecture for hardware implementation of the HOG algorithm. They speed up the histogram generation by reusing the histogram bins generated for the adjacent cell. 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 value. They speed up their design using this method and achieve 48 fps for 1920×1080 images.

Long et al. [78] 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 FASTCAM SA-X2. 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 the computations of the further steps.

Ngo et al. [91] propose a long pipeline architecture for the HOG algorithm with 155 stages. Although their proposed system contains a processor and the FPGA part for the HOG algorithm, 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. [63] 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 still their design 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 a higher speed of calculations. Naturally, they consume more hardware resources than the work 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 systems. In section 3.3.2, we will review the work based on hardware–software co-design of the HOG algorithm.

3.3.2 Hardware–software implementation of the HOG algorithm

In this work, 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. [14] 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 the speed while consuming more hardware resources. Their work is a hardware–software co-design, as they use CPU to control the pipeline of the HOG algorithm. They simplify the HOG computation by such methods as using the CORDIC algorithm for gradient calculation, using the Newton method for histogram normalization, and using specific bit-widths for different modules. They store the intermediate data of the histogram of the cells in SRAM memory and load the data for the further steps. Ma et al. [52] propose a hardware–software co-design approach for HOG-SVM computation. They profile the code on 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. [14], so as to have a more efficient implementation. Rettkowski et al. [60] 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 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 due to the software limitations, 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 higher frame rate of 39.6 fps. Huang et al. [71] 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 the classification step. 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. [64], 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. [59] 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), which is a variation of the SVM as 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 [14], 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 [52], [57], [60], [64], [71]. This can obviously become the bottleneck of the system, since the processor is usually slower than the programmable logic and processes data sequentially. In [59], [64], [71], 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 superior choice to increase the speed of the design. In this work, 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 control loops and address generation task to the processor. Therefore, the processor does not have negative impacts on the processing speed of the algorithm. In our design, we integrate feature extraction and classification in a unified pipeline to increase the speed of the process.

3.4 A novel hardware-software co-design of the HOG-SVM system

In this section, we propose a hardware–software co-design system for HOG implementation. As a case study of the HOG algorithm's application, we choose human detection, which is an online application. The INRIA person dataset [92] 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™) using Vivado®. 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 [15]. 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. [63]. 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 3.4. 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.

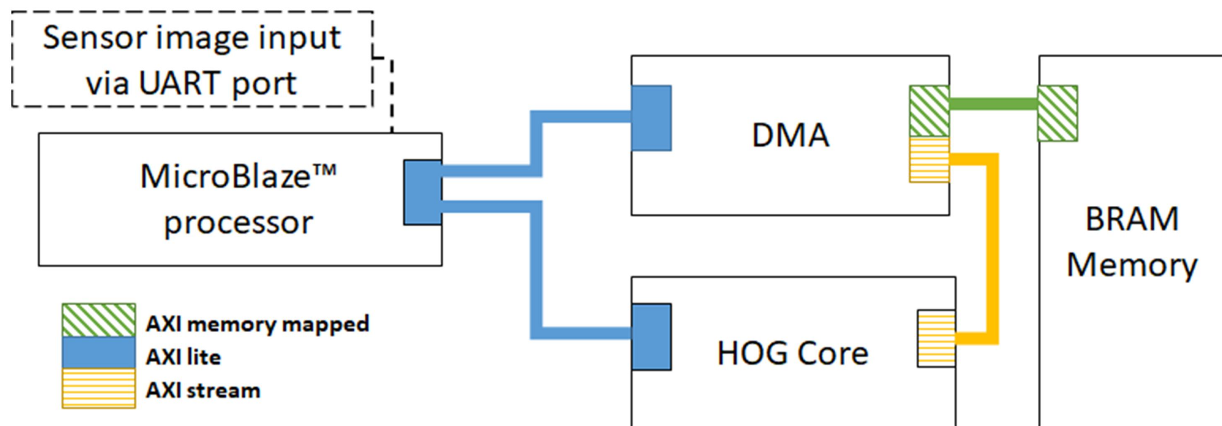

Figure 3.4: Block diagram of the proposed design and port connections

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.

3.5 HOG-SVM core

The overall diagram of the fully pipelined HOG-SVM implementation is shown in Figure 3.5. 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 3.5, 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.

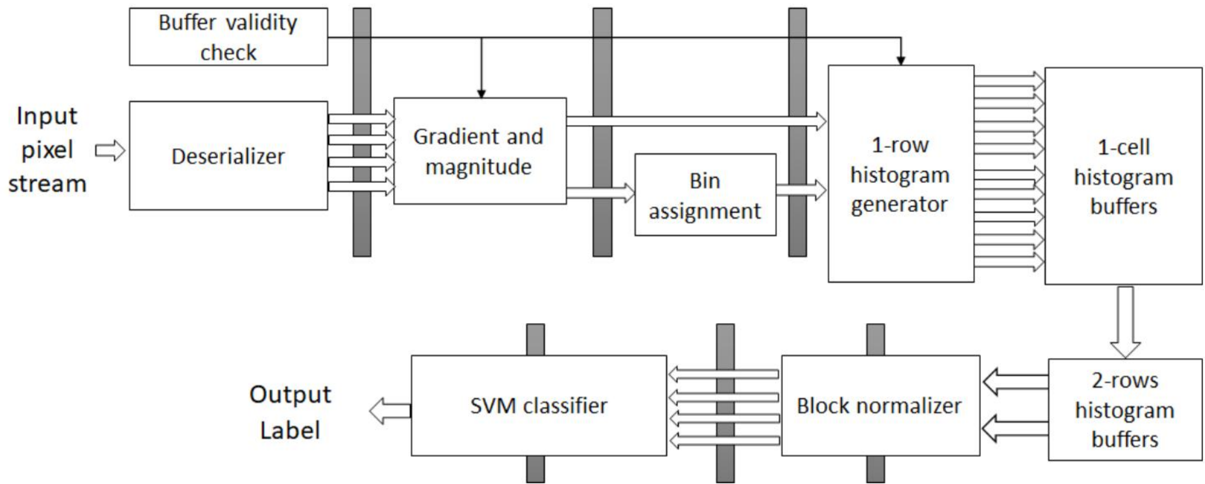

Figure 3.5: The overall diagram of the HOG-SVM core.

3.5.1 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 3.6 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×W clock cycles to fill all registers before producing valid outputs.

Figure 3.6: 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.

3.5.2 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 [38], the original HOG algorithm requires 2×W×H multiplication operations (for computing the square of the gradients twice for each pixel), W×H additions (once for each pixel), and W×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).

3.5.3 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 [81]. 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 (3.7)–(3.10) demonstrate the mathematical procedure of this method. Equation (3.7) presents the original orientation computation comparison. In the logarithm-based method, we first compute the tangent of all values as in Equation (3.8). Then, we compute the absolute value and then the base 2 logarithm to all values, as in Equation (3.9). 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 Equation (3.10). 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.

$$\theta_i < tan^{-1}(\frac{G_y}{G_x}) \leq \theta_{i+1} \tag{3.7}$$

$$tan(\theta_i) < \frac{G_y}{G_x} \leq tan(\theta_{i+1}) \tag{3.8}$$

$$log_2(|tan(\theta_i)|) < log_2(|\frac{G_y}{G_x}|) \leq log_2(|tan(\theta_{i+1})|) \tag{3.9}$$

$$log_2(|tan(\theta_i)|) < log_2(|G_y|) - log_2(|G_x|) \leq log_2(|tan(\theta_{i+1})|) \tag{3.10}$$

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 3.6 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.

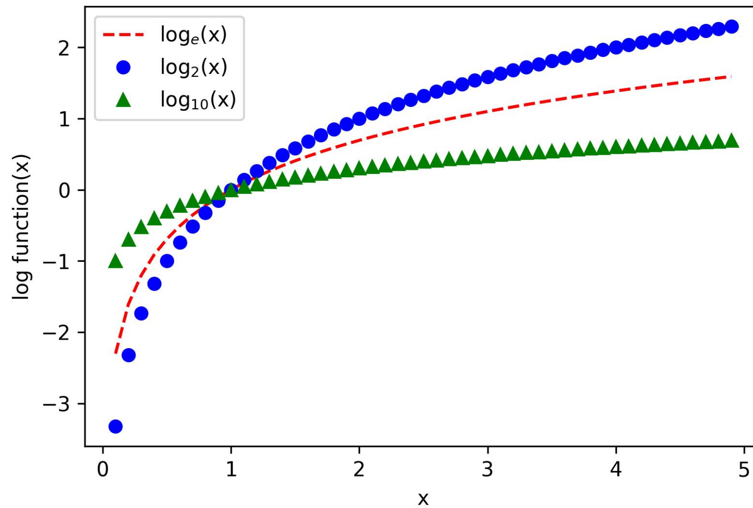

Figure 3.7: Difference between the slope of three logarithm functions.

To compute the values, we use an LUT-based (Look Up Table) RAM. Depending on the input value

Table 3.1: Limits for bin assignment

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

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 (3.11) shows the scaled version of (3.10). Both sides of the inequality are precomputed, and for the middle expression, one addition to 160 is added. The reason for adding 160 is that we multiply all sides of (3.8) 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.

$$32log_2(32|tan(\theta_i)|) < 160 + 32log_2(|G_y|) - 32log_2(|G_x|) \leq 32log_2(32|tan(\theta_{i+1})|) \tag{3.11}$$

After that, the appropriate bin is selected using the sign bit, as in Figure 3.8. 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 3.8, v is the term computed by subtraction of the logarithm values. Depending on the sign bit, a range of the orientations is chosen.

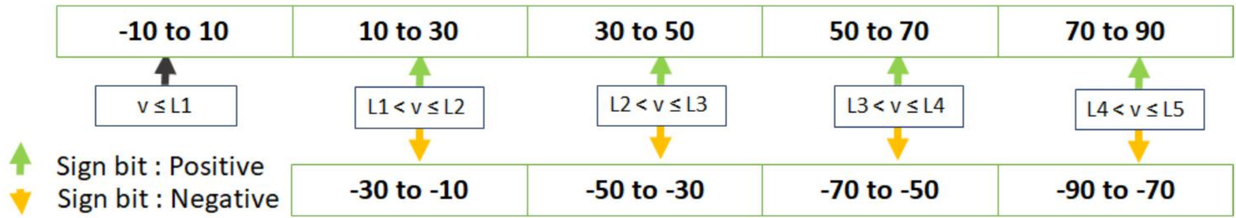

Figure 3.8: The bin assignment procedure.

The limit values in Figure 3.7 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 [38], in the original HOG algorithm, the orientation and bin assignment module require W×H arctangent operations, W×H divisions, and 9×W×H comparison operations. Although some previous works [81] 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

Algorithm 1 The pseudo-code for the bin assignment step

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

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.

3.5.4 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 3.9, 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 3.10 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.

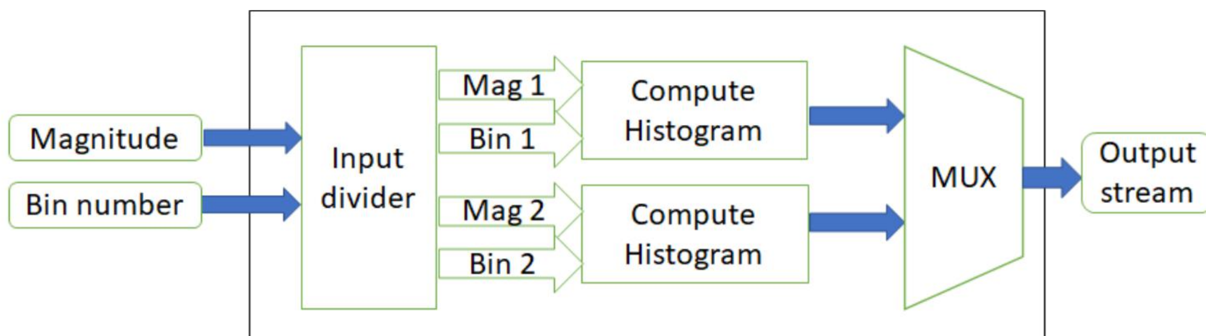

Figure 3.9: One-row histogram generation module.

Figure 3.10: The time-sharing protocol for the histogram generation module.

3.5.5 One-cell histogram buffers

We employ the same architecture proposed by Luo et al. [63] 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 3.11 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.

3.5.6 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 3.12 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×W/8 clock cycles to generate the first valid output, since there are two rows and we have W/8 registers in each row.

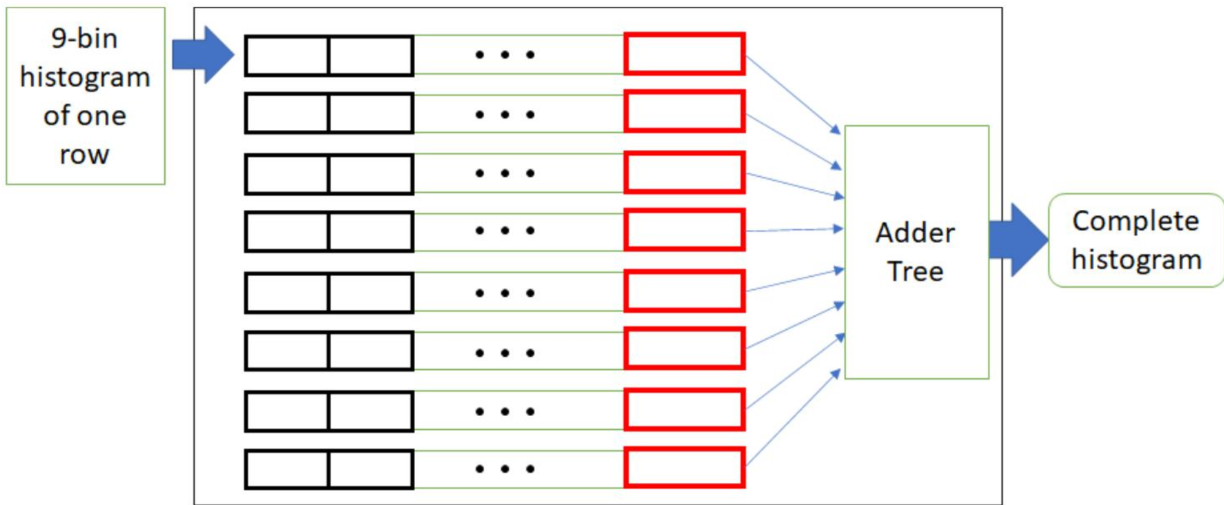

Figure 3.11: The block diagram of one-cell histogram buffers.

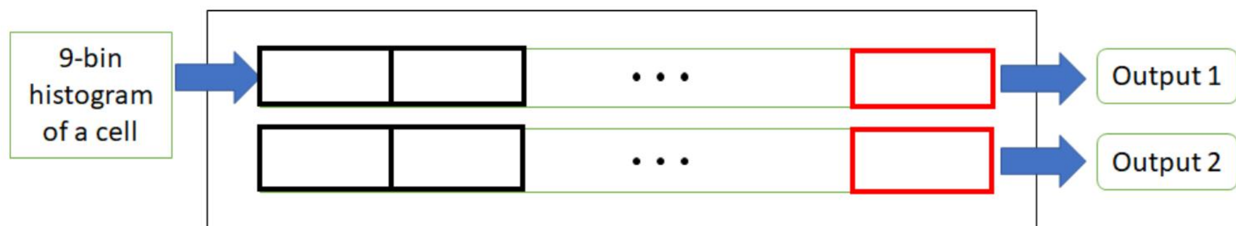

Figure 3.12: The block diagram of two-row histogram buffers.

3.5.7 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 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 3.13 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

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

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.

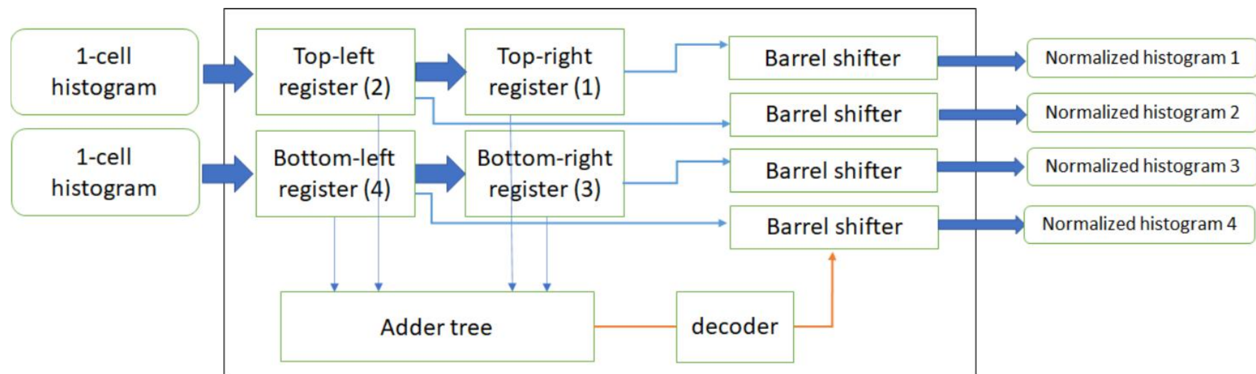

Figure 3.13: The block diagram of block normalization module.

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 [38] 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).

3.5.8 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 3.14.

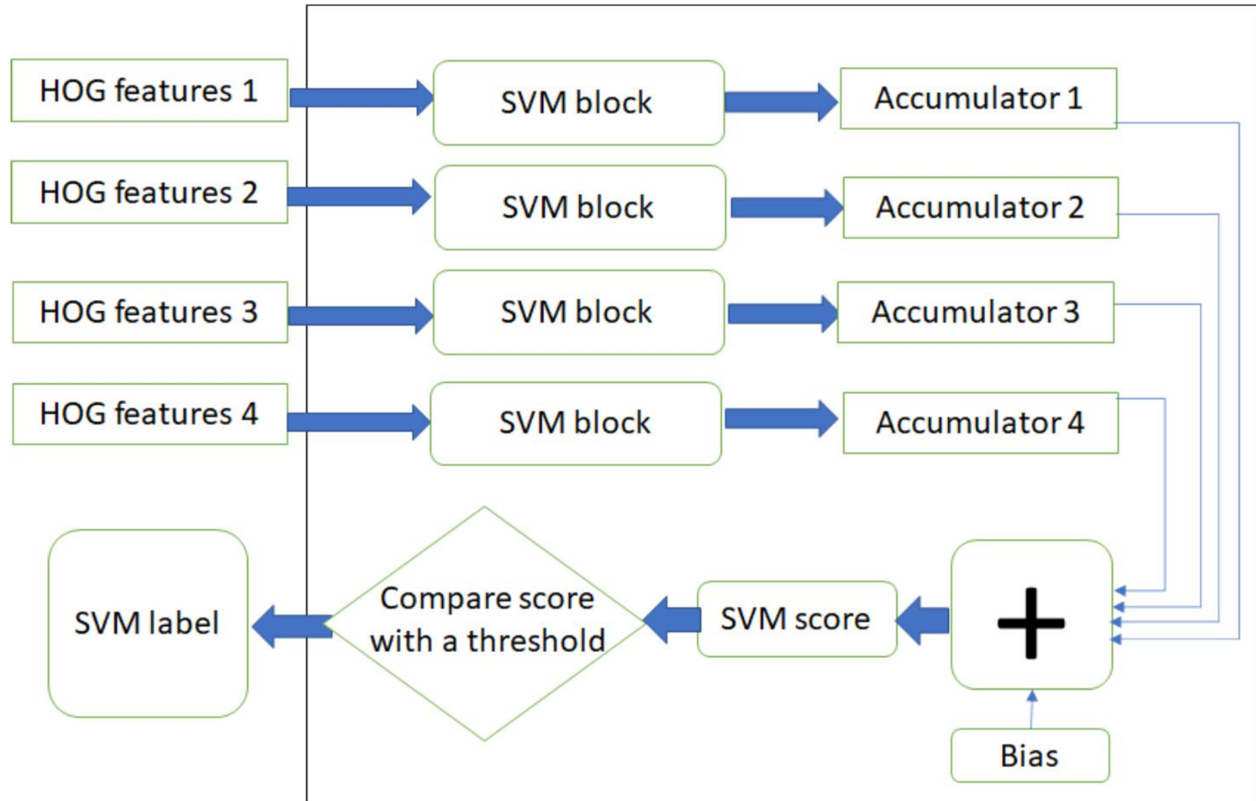

Figure 3.14: The SVM classifier module.

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 3.15. 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.

3.6 Results and comparison with other work

Rettkowski et al. [60] 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

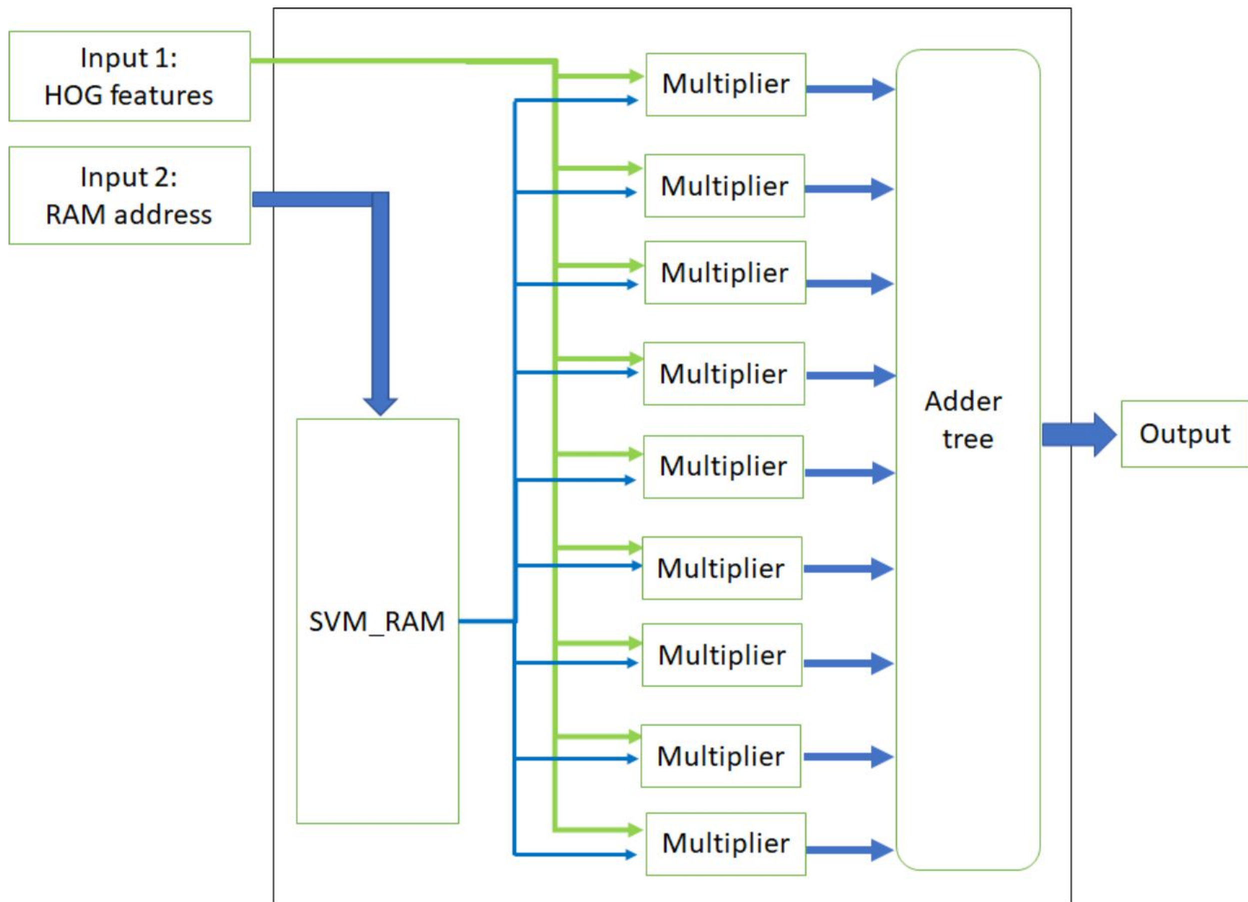

Figure 3.15: SVM block internal logic.

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 [52], [57], [59], [60], [64], [71], 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 [14]. Intermediate communications with external off-chip memory 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 [59] 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

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. [60]*	Zynq®	1920×1080	41,858	1584	13	39.6	0.99
	Ngo et al. [91]	Cyclone® V	640×480	13,646	317	38	75	0.46
	Long et al. [78]	Stratix® IV	512×512	266,023	47	236	2500	8.19
	Luo et al. [63]	Cyclone® IV	800×600	16,060	334	69	162	0.51
	Qasaimeh et al. [65]	Zynq®	1920×1080	32,871	NA	130	48	0.59
Hardware–Software Co-design	Mizuno et al. [14]	Cyclone® IV	800×600	34,403	334	68	72	0.86
	Ma et al. [52]	Virtex®-6	640×480	184,953	13737	190	68	0.14
	Bilal et al. [59]	Cyclone® IV	640×480	65,501	103	10	25	0.15
	Yu et al. [57]	Spartan®-6	640×480	15,167	351	19	1.5	NA
	Rettkowski et al. [60]*	Zynq®	350×175	NA	NA	NA	0.44	0.0001
	Ngo et al. [64]	Cyclone® V	640×480	12,138	437	65	11	0.02
	Hunag et al. [71]	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.

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. [78] has the highest pixel per clock cycle value. The reason is that in [78], 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. [14], 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. [60] use a Zynq® family FPGA, while Ma et al. [52] use a Virtex® series FPGA. Mizuno et al. [14], Bilal et al. [59] and Ngo et al. [64] 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. [52] 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

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.

window in one direction. Figure 3.16 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.

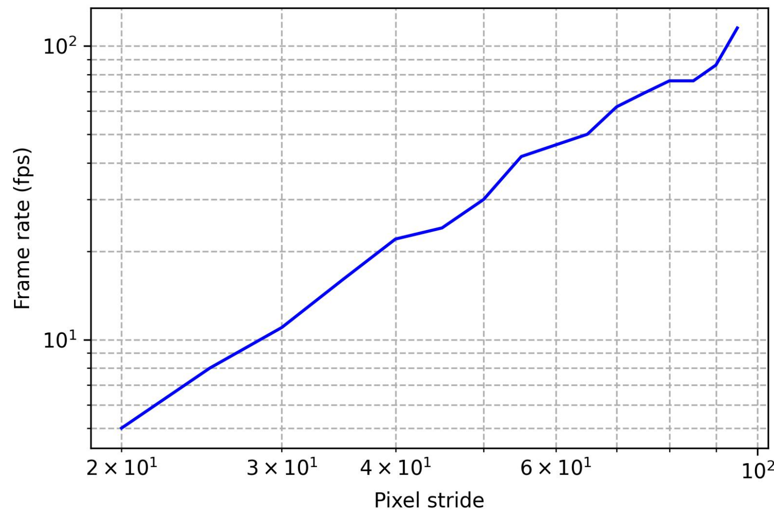

Figure 3.16: The relation between pixel stride and frame rate.

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.

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.

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.

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 3.5, 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. [63]. The accuracy results of our system are shown in Figure 3.17. 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 3.17 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 [15].

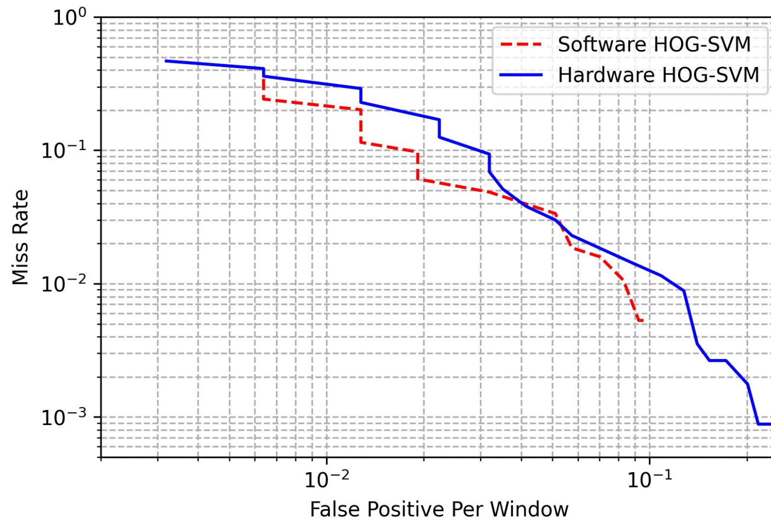

Figure 3.17: Comparison of software implementation and our proposed HW-SW co-design.

3.7 Conclusions

In this work, 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 [93], which require a high number of computations and can benefit from parallel implementation.

Chapter 4

A Fully Pipelined FPGA Architecture for Multiscale BRISK Descriptors With a Novel Hardware-Aware Sampling Pattern

Binary descriptors have been shown to be faster than non-binary descriptors while producing comparable results in image matching applications. In recent years, there have been many attempts to design hardware accelerators for extraction of binary descriptors to achieve higher processing rates. One of the well-known methods is the Binary Robust Invariant Scalable Keypoints (BRISK) algorithm, which has shown outstanding results in various applications. In this work, we propose a multi-scale FPGA-based hardware architecture for the BRISK descriptor. In addition, a new image sampling pattern for the BRISK algorithm is described which is shown to be more efficient than the original sampling pattern for hardware implementation. Our new sampling pattern decreases the size of the patches containing the keypoints to one quarter of the size of that used in the original BRISK algorithm, which leads to a reduction in FPGA resource utilization while maintaining the accuracy of the image matching application. Our proposed design is fully pipelined and achieves a frame rate of 78 fps on images with full HD resolution.

4.1 Introduction

Keypoint detection and description has many applications in computer vision. Keypoint detection is the process of finding the location of keypoints (or interest points), which are the points in the image such as corner features that represent important information. Extracting features from a patch (a small window of image which is being processed by the descriptor) around the keypoint is called keypoint description. Features are any information from the patch that can be used for specifying each keypoint individually. Extracted features should have high similarity for a keypoint which is visible in two different images while having low similarity with the features of other keypoints in the same image and other images. Invariance

to illumination, scale, and rotation are important characteristics of feature descriptors.

There are many feature detection and description algorithms proposed in the literature. These algorithms are commonly categorized into two groups. The first one is non-binary descriptors including SIFT [19], SURF [16], and HOG [15]. The second category is binary descriptors including BRIEF [20], FREAK [21], BRISK [22] and ORB [12]. Non-binary descriptors usually generate histograms of image features such as gradients and use them for describing a patch of an image. On the other hand, binary descriptors are commonly based on the comparison of the intensity of different pairs of pixels in a patch around a keypoint. Since non-binary descriptors process more information, they typically produce more accurate results. However, the main advantage of binary feature descriptors over non-binary ones is faster computation while maintaining comparable accuracy.

4.1.1 Hardware implementation of descriptors

Although binary descriptors have been shown to produce a noticeable performance enhancement in terms of speed compared to non-binary descriptors, they remain computationally expensive. This has led many researchers to work on hardware implementation of binary descriptors to achieve higher speed. Implementing hardware accelerators can make computer vision applications more practical due to the benefits of processing multiple computations in parallel. Hardware accelerators do not have the limitations of processors with conventional architectures. In particular, Field Programmable Gate Arrays (FPGAs) are popular platforms for implementing hardware accelerators for their ease of implementation and reasonable time-to-market in comparison with application specific integrated circuits. There are many attempts to implement descriptor algorithms such as HOG, ORB, and FREAK on FPGAs due to their low power consumption and parallel computation capabilities. As an example, an analysis of FPGA-based implementation of the HOG algorithm is provided in our previous work [38].

4.1.2 BRISK algorithm

The BRISK algorithm combines the AGAST detector [94] and a new descriptor [22]. Since the focus of our work is on the description portion of the BRISK algorithm, we briefly introduce the description process of BRISK in this section.

When a keypoint is detected in an image, a patch of pixels around that keypoint is extracted. Then, specific locations of pixels around the keypoints are used as samples of the patch to generate the descriptors. The original BRISK algorithm uses a sampling pattern as shown in Fig. 4.1 [22].

The sampled pixels around a keypoint in a patch form groups as pairs and depending on the distance between each sample in a pair, are labeled as a *long distance pair* or a *short distance pair* [22]. If the distance is larger than a specific threshold, that pair is labeled as a long pair. Otherwise, it is labeled as a short pair. Figure 4.2 shows examples of long pairs and short pairs. In the original BRISK algorithm, 870 long pairs and 512 short pairs are selected from the 60 samples including the keypoint. After selecting the long pairs, they calculate the orientation of the patch. The first step is to compute the gradient of each long pair as in (4.1):

$$g(P_i, P_j) = \frac{P_i - P_j}{\|P_i - P_j\|} \times \frac{I(P_i, \sigma_i) - I(P_j, \sigma_j)}{\|P_i - P_j\|} \tag{4.1}$$

where P_i and P_j are the coordinates of sample points in each pair and $I(P_i, \sigma_i)$ is the intensity of an image

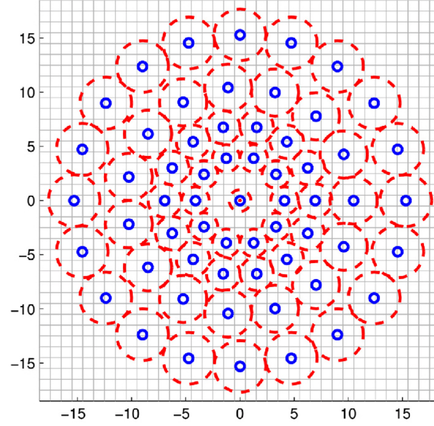

Figure 4.1: Original BRISK sample points [22] (©[2011] IEEE).] The blue circle in the center represents the keypoint. Other blue circles represent sample points. The red dashed circles represent the relative variance of the Gaussian filter which is applied around each sample point. The keypoint is positioned on (0,0) and the numbers on the vertical and horizontal axes represent the relative position from the origin, in pixels.

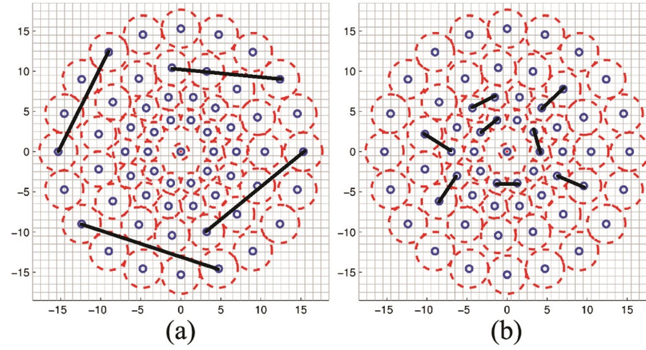

Figure 4.2: Examples of long pairs (left image) and short pairs (right image) based on the BRISK sample pattern [22] (©[2011] IEEE).

smoothed by a Gaussian kernel with variance of σ_i in location P_i. After computing the gradient for each pair, the summation of all the gradients of long pairs is computed in horizontal and vertical directions together as in (4.2):

$$G = \begin{pmatrix} g(x) \\ g(y) \end{pmatrix} = \frac{1}{L} \sum_{P_i, P_j \in patch} g(P_i, P_j) \tag{4.2}$$

Finally, the patch orientation is determined by computing the arctangent of G as in (4.3):

$$\theta = arctan(\frac{g(y)}{g(x)}) \tag{4.3}$$

where θ is the orientation of the patch. Each patch has its own main orientation, which represents the main gradient direction of that patch. If we rotate all the patches so that their main orientation is in one direction, then the patches of the same keypoint in two different images with different orientations will transform into the same direction. Therefore, in the next step, the algorithm rotates each patch by its orientation. Then,

BRISK compares the intensity value of each pixel from short pair samples with the other pixel in the same pair. Instead of comparing the raw intensity of the pixels, they smooth the image using the pixels around it. Smoothing has a significant effect on the accuracy of the algorithm. If the smoothed value of the first pixel is greater than the second one, they set the corresponding bit in the descriptor to one and otherwise they set it to zero.

4.1.3 New approaches to enhance acceleration

There has been little publication in the academic literature focusing on implementation of the BRISK algorithm on FPGAs, particularly at multiple scales. In this work, we propose and evaluate a design of a multi-scale hardware implementation of the BRISK algorithm to achieve scale invariant performance.

We propose a novel design of a multi-scale pipeline architecture, including innovations in various stages of the pipeline. Scale invariant descriptors can match objects of various sizes in images. There are two conventional methods for implementing multiple scales of descriptors in hardware. The first method is to extract descriptors from multiple frame sizes sequentially, as done by Liu et al. [95] for the ORB descriptor. The second method is to replicate the buffers and computational components for the number of scales to process multiple scales in parallel as used by Sun et al. [33]. The second method is faster but requires more hardware resources in comparison with the first method.

Our multi-scale hardware implementation of the BRISK algorithm requires less hardware resources than trivial replication of the circuits for single scale. In particular, we store and process a quarter of the resized image in each scale without loss of accuracy. The pipeline architecture in our design is synchronized so that we can share one of the key computation stages of the algorithm between multiple scales.

We also introduce an innovative hardware-aware sampling pattern which is designed based on minimization of hardware resources, which facilitates the implementation of the BRISK algorithm for multiple scales. Using this sampling pattern, we process one quarter instead of the full image data in each scale while maintaining comparable accuracy. We demonstrate a fully-pipeline architecture for this algorithm which has extensive parallelism in each stage of the pipeline. In addition, we use a variety of techniques such as resource sharing, clock gating, and computational approximations to make our design more efficient in terms of power and resource utilization.

4.1.4 Organization of the paper

The remainder of this paper is presented as follows. In section II, we compare well-known binary descriptors and discuss recently published FPGA implementations of these algorithms. In section III, we introduce our new hardware-aware sampling pattern for the BRISK algorithm. In section IV, we present a novel multi-scale hardware implementation of the BRISK algorithm. In that section, we provide our hardware design and solutions in detail for each part of the BRISK algorithm. We provide a comprehensive analysis of the benefits of our contributions in section V and evaluate our innovations in comparison with recently published, state-of-the-art results. Finally, we conclude this paper in section VI.

Table 4.1: A comparison of examples of binary descriptors

Descriptor algorithm	Original detector	Scale invariance	Sampling method	Rotation computation method
BRIEF [20]	CenSure [1]	No	Random pairs of pixels in the patch	—
FREAK [21]	Multi-scale AGAST [94]	Yes	Predefined samples for orientation and description	Main direction computed based on predefined samples
BRISK [22]	Multi-scale AGAST [94]	Yes	Long pairs for orientation, short pairs for description	Main direction computed based on long-pair samples
ORB [12]	FAST [10]	Yes	Random pairs of pixels in the patch	Main direction computed based on intensity centroid of the patch

4.2 Related work

In this section, we provide a comparison of commonly-known binary descriptors. We continue this section by reviewing recent advances in hardware implementation of binary descriptors then focus on work which implements the BRISK algorithm.

4.2.1 A comparison of binary descriptor algorithms

Although binary descriptors operate similarly in that they use comparison for generating their output, there are important differences in the associated algorithms. Table 4.1 shows the main differences in sampling and rotation computation among four well-known binary feature descriptors. In this work, we propose a new design for the BRISK descriptor since it has many applications in various specialized computer vision fields such as bone age assessment [96], SAR-based automatic target recognition [97], content-based image retrieval [98], and emotion recognition [99]. It requires fewer computations than non-binary descriptors such as SIFT and SURF while it produces comparable results. The FREAK descriptor is very similar to BRISK with the difference between them being the sampling pattern and rotation invariant calculation. The samples in FREAK are mostly focused on the center of the patch while BRISK has a more uniformly distributed pattern throughout the patch. Therefore, for various applications they produce close but different results. BRISK achieves better results than BRIEF since it is rotation and scale invariant. Also, since the original BRISK generates 512-bit descriptors while ORB generates 256-bit descriptors, it has been shown to outperform the original ORB in specific applications [23]. However, BRISK is more computationally demanding since it processes a larger number of pixels from the same patch size. Mouats et al. [100] evaluate BRISK, FREAK, ORB, and non-binary descriptors in poor lighting conditions. In most test cases, BRISK outperforms other binary descriptors in accuracy.

Binary descriptors are faster than non-binary descriptors since they provide a binary vector based on the comparisons of pixel values in the final feature vector, while non-binary descriptors normally have more complex computations such as histogram generation and dense gradient computation. Speed comparison of descriptors have been addressed in previous work [11], [7], and [101]. However, a descriptor such as BRISK will still require a high number of computations. In the original work, for each keypoint, the algorithm

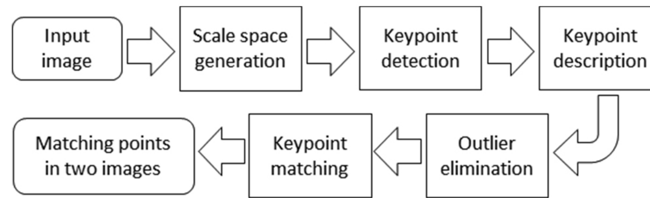

Figure 4.3: Image matching pipeline.

requires 870 long-pair computations and 512 short-pair comparisons which are processed sequentially on a conventional CPU.

In our work, we propose an FPGA-based design to compute the descriptors faster in multiple scales by performing the calculation of long pairs and short pairs in parallel. Multi-scale description results in a more accurate image matching system.

4.2.2 FPGA-based implementations of binary descriptors

There are many FPGA-based implementations of binary descriptors. Sun et al. [102][33], Lima et al. [103], Liu et al. [95], and Tran et al. [104] implement the ORB algorithm on hardware. Fang et al. [105] implement ORB algorithm on an FPGA for two scales of 640×480 and 533×400. They stop the streaming input whenever a keypoint enters the line-buffers so that the descriptor has enough time to calculate the features. This design choice leads to lower frame rate. They achieve a frame rate of 67 fps with maximum frequency of 203 MHz for 640×480 resolution. Kalms et al. [106] and Kapela et al. [107] propose hardware architectures for the FREAK descriptor on an FPGA and Pham et al. [108] design an FPGA-based architecture for rotation-aware BRIEF algorithm. Although the BRIEF algorithm requires less calculation than BRISK, they achieve 60 fps for 1920×1080 images. For comparison, in our work, we achieve 78 fps on 1920×1080 images.

4.2.3 FPGA-based implementations of BRISK descriptor

Despite the large number of computations for orientation compensation, there has been little work focusing on hardware implementation of the BRISK algorithm. Ulusel et al. [108] implement the BRISK algorithm on an FPGA. They implement a single-scale version of the BRISK algorithm with an 800×480 image resolution. Their work does not contain details of orientation compensation which is an important part of the description. They also do not provide any results for demonstrating accuracy. Azimi et al. [35] propose a fully pipelined and parallel hardware architecture for the detector part of the BRISK algorithm which is a multi-scale FAST [10] algorithm. Since their focus is only on detection, they have not implemented the descriptor part of BRISK.

A complete image matching system which includes scale space generation, keypoint detection, keypoint description, keypoint matching, and outlier elimination is shown in Fig. 4.3. In this work, we focus on the descriptor and implement it with a hardware accelerator. We also implement a FAST detector as a part of the matching system. However, the descriptor could be combined with any feature detector unit whether implemented in software or hardware.

Table 4.2: Effect of changing the distance threshold T on the number of sample points with n=36 (θ=10)

Distance threshold T	Number of sample points
0.98	21
1.00	57
1.02	65
1.04	73

4.3 Hardware-aware sampling pattern for BRISK

One of the key differences among binary descriptors is their image sampling pattern. In this section, we propose a new, novel sampling pattern which is more efficient in resource utilization for hardware implementation than the sampling pattern of the original BRISK algorithm. The main idea is to constrain the sampling points to be only in even (or odd) rows and columns. We achieve similar accuracy with patterns in even coordinates, while reducing the processing requirements to one quarter of the original algorithm.

Figure 4.4 presents examples selected from the sequence of steps for generating the proposed sampling pattern. Similar to BRISK, our proposed sampling pattern is based on a 33×33 patch. To obtain this pattern, we start with a grid of 17×17 pixel locations. First, all the pixels around the center of the grid are selected as initial sampling points as shown in step 0 of Fig. 4.4. Then, for a specific number of rotations n, we determine the angle step θ based on (4.4).

$$\theta = \frac{360}{n} \tag{4.4}$$

We rotate the grid for every θ=10 degrees (n=36) and compute the Euclidean distance of the center of each rotated pixel with the nearest unrotated ones. If this distance is less than a predefined threshold T, we mark it as an acceptable overlap. For each degree, we calculate the acceptable overlaps between samples. The samples that do not have acceptable overlaps are removed as shown in the examples of Fig. 4.4. Last, we have the intersection of all acceptable overlaps for all rotations. The resulting samples are the pixels which have acceptable overlaps with each other in all angle steps θ for the specified threshold. If we reduce the threshold, the number of acceptable overlaps becomes smaller, which is shown in Table 4.2. Higher thresholds lead to a greater number of acceptable overlaps. However, if we accept overlaps with higher distances it will decrease the accuracy of matching since the matching algorithm would assume those pixels have the same position in rotated images. As shown in Table 4.3, decreasing the angle step θ leads to more precise rotation of the pixels. Therefore, the acceptable overlaps are selected from the intersection of more pixel rotations and the number of pixels which are available in all possible rotations decreases. As a consequence, the number of acceptable overlaps decreases.

Finally, we multiply each coordinate by two so that the samples would be extracted from only even rows and columns of a 33×33 patch. Figure 4.5 shows examples of sampling patterns using different thresholds. Figure 4.6 presents different sampling patterns obtained from different angle steps.

We choose a threshold of 1 pixel and 36 angle steps for the sampling pattern. This means that we have samples for each 10 degrees of rotation and the sample points are the ones which have the shortest distance to at least one sample point in another rotation of the same sampling pattern. This design choice gives us enough samples to describe a patch with adequate accuracy but not so many samples such that FPGA

Table 4.3: Effect of changing the number of rotations on the number of sample points with T=1

Number of rotations n	Angle step θ	Number of sample points
1	360	289
12	30	169
24	15	89
36	10	57
48	7.5	33
60	6	1

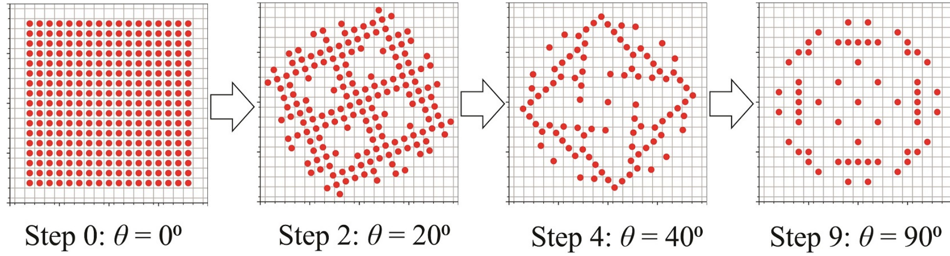

Figure 4.4: Examples selected from the sequence of steps for generating the proposed sampling pattern (with T=1, n=36). The initial sampling points are shown as step 0. Each step rotates the sampling points by θ=10 and the samples without acceptable overlap are removed. Not all steps are shown for brevity.

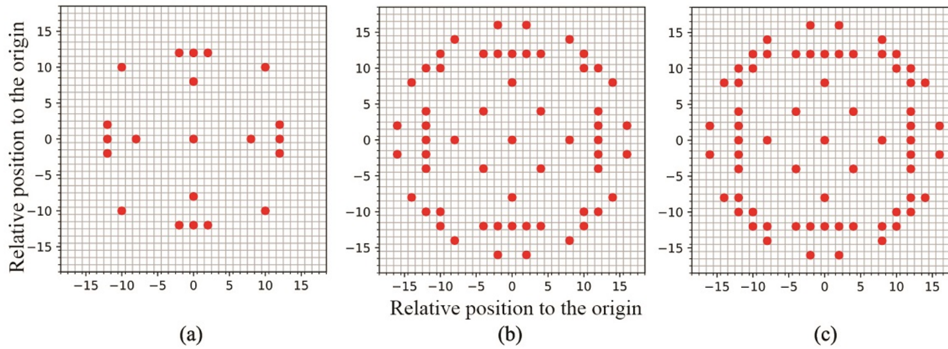

Figure 4.5: Examples of sampling patterns resulting from changing threshold (T) to produce different sampling patterns for the same number of rotations n. For all three patterns n=36 (θ=10). (a) is for T=0.98, (b) is for T=1, and (c) is for T=1.02.

routing is unnecessarily complicated.

4.4 Hardware implementation of the multi-scale BRISK algorithm

In this section, we introduce the architecture of our design for the BRISK algorithm implementation. The proposed design is a fully pipelined architecture so that at each clock cycle, a new pixel enters the pipeline without a need to pause. In this design, our main focus is first, to achieve higher speed by parallelizing the computations required in each step of the algorithm and second, to reduce hardware resource usage by using our novel sampling pattern. The proposed architecture is shown in Fig. 4.7. The controller is a combination

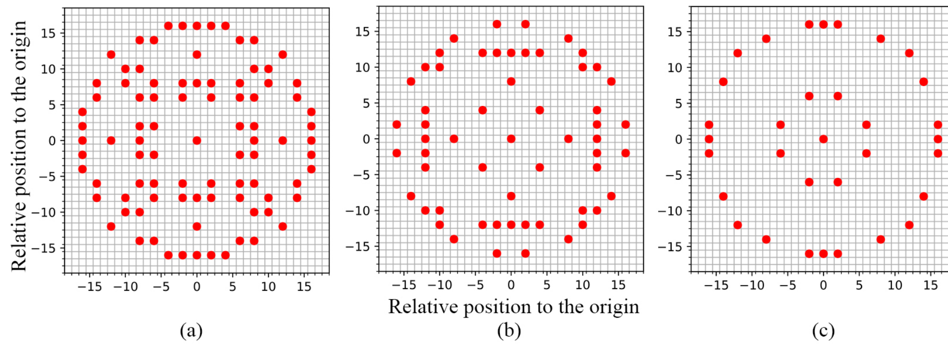

Figure 4.6: Examples of sampling patterns resulting from changing the number of rotations n to produce different sampling patterns for the same threshold T. For all three patterns T=1. (a) is for n=24 (θ=15), (b) is for n=36 (θ=10), and (c) is for n=48 (θ=7.5).

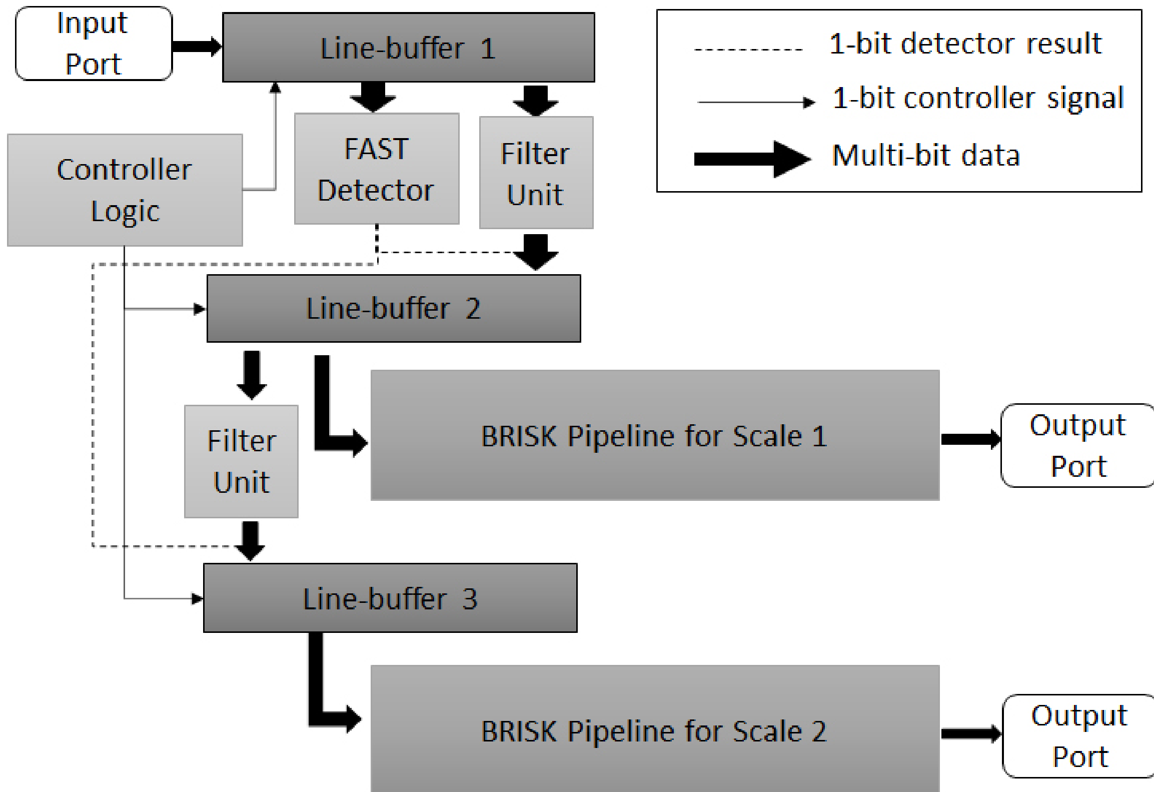

Figure 4.7: The overall architecture of our design. This design includes two parallel pipeline paths for two scales. The controller logic is a combination of multiple finite state machines. The dashed line is a single-bit result of the detector which is concatenated with the pixel values.

of finite state machines (FSMs). We discuss the multi-scale structure of this architecture in more detail in section IV B. The streaming input pixel enters the design one pixel at each clock cycle. The pixels enter a line-buffer which has a size of $W\times 11$ (where W is the width of the image). After an initial waiting phase, the line-buffer becomes full and the output of the line-buffer which is an 11×11 patch will have valid values.

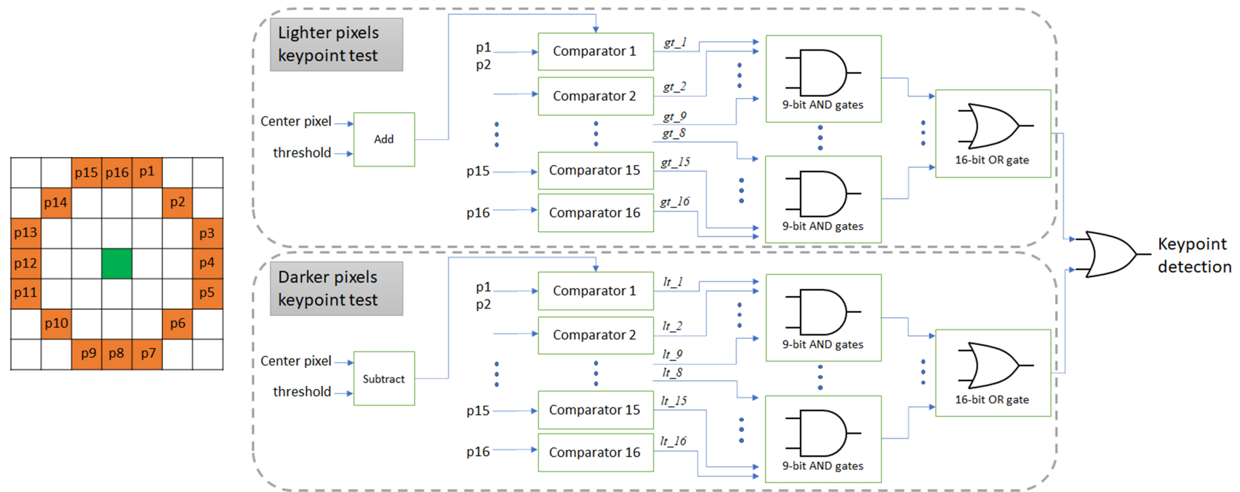

Figure 4.8: Architecture of the FAST keypoint detector. The window on the left shows the location of the 16 pixels around the center pixel. The two keypoint tests process the pixel values in parallel.

We use the 11×11 patch at the end of the line-buffer as an input for the FAST detector and the filter unit.

The FAST detector tests each 11×11 patch to determine if the center pixel of the patch is a keypoint. This module has two main parts, the lighter pixels keypoint test and the darker pixels keypoint test. These two parts process the pixels in parallel. First, we extract the values of the 16 pixels in a 7×7 patch around the center pixel shaping a Bresenham circle as shown in Fig. 4.8. We define an upper limit and a lower limit which specify a neighborhood around the center pixel value. We calculate the upper limit by adding the value of the center pixel to a predefined threshold, and at the same time, we calculate the lower limit by subtracting the center pixel from the same threshold. This threshold is used for reducing the effect of noise on detection result. We choose the value of 10 for this threshold based on our experiments. The upper limit is used for the lighter pixels test and the lower limit is used for the darker pixels test.

For the lighter pixels test, we compare all 16 pixels around the center pixel with the upper limit simultaneously to see if 9 consecutive pixels are lighter than the center pixel. We use logic gates to AND the "greater than" output of every 9 consecutive comparators. If any one of the AND operations result is true, the center pixel is identified as a keypoint. We logically OR the results of the AND operations. The result of the OR gates is a 1-bit keypoint detection output.

Simultaneously, we compare the pixels around the center pixel with the lower limit to check the criterion for darker pixels. For darker pixels, we check the "less than" output of the comparators. Finally, we OR the result of the darker and lighter keypoint detection to form the output of the detector module. Since we have 16 comparators and 16 9-bit AND gates, 8 of the inputs of each two adjacent AND gates are common between them. Note that in Fig. 4.8, signals *gt_1* to *gt_9* are connected to the top AND gate and signals *gt_8* to *gt_16* are connected to the bottom AND gate as an example. In this example, *gt_8* and *gt_9* are common inputs of these two gates.

The filter unit contains a constant weight window that has higher values in the center and lower values towards the borders. The highest value in the center is 1 and the lowest value at the borders is 0.5 as shown in Fig. 4.9. Due to the focus on the pixel in the center and keeping the effect of the surrounding pixels, these weights lead to an acceptable accuracy for our design. We multiply each pixel in the input patch with

Table 4.4: Approximations of multiplications in the filter unit

Desired multiplier	Approximated multiplier	n_i
0.9	0.90625	1, 2, 3, 5
0.8	0.796875	1, 2, 5, 6
0.7	0.695312	1, 3, 4, 7
0.6	0.6015625	1, 4, 5, 7
0.5	0.5	1

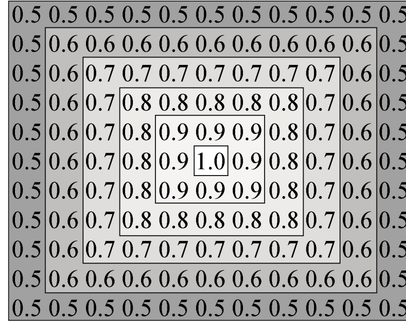

Figure 4.9: Weights of the filter unit that is shown in Fig. 4.7.

the corresponding value of the filter patch. Since the weights are constants, we use logical shift and adder logic. We approximate each multiplication by logical shift and adder logic with four terms as shown in (4.5).

$$M_{i,approx} = \sum_{j=1}^{4} M_i \gg n_i(j) = \sum_{j=1}^{4} M_i \times (\frac{1}{2^{n_i(j)}}) \tag{4.5}$$

In (4.5), M_i is the multiplicand, $M_{i,approx}$ is the approximated multiplicand, and n_i is an array of four values. The values for each $n_i(j)$ is selected so that the difference between M_i and $M_{i,approx}$ is acceptable. As an example, for appoximating 0.9, we can use $n_i = 1, 2, 3, 5$ which results in $M_{i,approx} = 0.90625$. Table 4.4 shows the values of each n_i and the approximations we use in this stage. Subsequently, we compute the summation of weighted pixels using a 7-level adder tree. Finally, we divide the output by the summation of all weights in the weight window in Fig. 4.9. Since this value is a constant, we use right shift and adder logic instead of division to approximate this value, which produces a negligible error.

The filter unit and the detector unit work in parallel. We concatenate the detector result (which is one bit indicating if the pixel is a keypoint or not) to the 8-bit value of the pixel. The result, which is a 9-bit value, enters the second line-buffer which contains the smoothed values of the pixels in the image. The length of the second line-buffer is half of the image width. By using even rows and columns, we do not lose any information since our sampling pattern is located on the pixel locations that remain in this line buffer. The advantage of this technique is that we can consider only one quarter of the pixels in the image from this step forward since the sampling pattern does not require pixel values from adjacent locations. The output of the second line-buffer is a 17×17 patch which can be accessed when this line-buffer becomes full of valid data. We send this patch to the first step of the BRISK pipeline which is the long-pair subtraction unit.

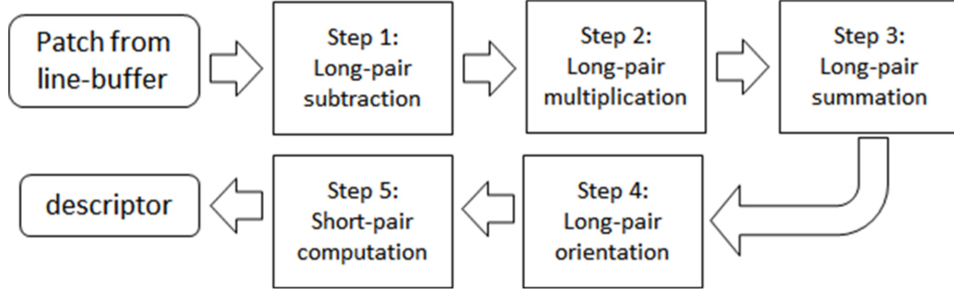

Figure 4.10: The block diagram of a pipeline architecture for one scale.

4.4.1 BRISK pipeline

The steps for BRISK pipeline include long-pair subtraction, multiplication, summation, orientation calculation, and short-pair comparison. In the proposed design, these stages are separated using registers to form a pipeline architecture. Therefore, at each clock cycle, each stage processes the data of a specific patch while the previous stage is processing the adjacent patch on the left. Figure 4.10 shows the block diagram of the architecture of the pipeline for one scale in our design. We use the detector bit (the 9th bit) of the output value of the line-buffer as an enable signal for the pipeline stages. Therefore, if the pixel entering the pipeline architecture is not a keypoint, the pipeline will not continue to work and we save dynamic power. We discuss the advantages of this method in section V. In this design, we compute the gradients in horizontal and vertical directions in a different order than the original BRISK algorithm. Since in the hardware implementation the coordinates of the samples are known, we can factor the coordinate terms and precompute some parts of the expressions in (4.1) and (4.2), and use them as coefficients as in (4.6) and (4.7):

$$\begin{aligned} g(x) &= \sum_{P_i, P_j \in patch} \frac{1}{L} \frac{(P_i(x) - P_j(x))}{\|P_i - P_j\|^2} (I(P_i, \sigma_i) - I(P_j, \sigma_j)) \\ &= \sum_{k=1}^{870} c_1(k)(I(P_{k,1}, \sigma) - I(P_{k,2}, \sigma)) \end{aligned} \tag{4.6}$$

$$\begin{aligned} g(y) &= \sum_{P_i, P_j \in patch} \frac{1}{L} \frac{(P_i(y) - P_j(y))}{\|P_i - P_j\|^2} (I(P_i, \sigma_i) - I(P_j, \sigma_j)) \\ &= \sum_{k=1}^{870} c_2(k)(I(P_{k,1}, \sigma) - I(P_{k,2}, \sigma)) \end{aligned} \tag{4.7}$$

where

$$\begin{aligned} c_1(k) &= \frac{1}{L} \frac{P_{k,1}(x) - P_{k,2}(x)}{|P_{k,1} - P_{k,2}|^2}, \\ c_2(k) &= \frac{1}{L} \frac{P_{k,1}(y) - P_{k,2}(y)}{|P_{k,1} - P_{k,2}|^2} \end{aligned} \tag{4.8}$$

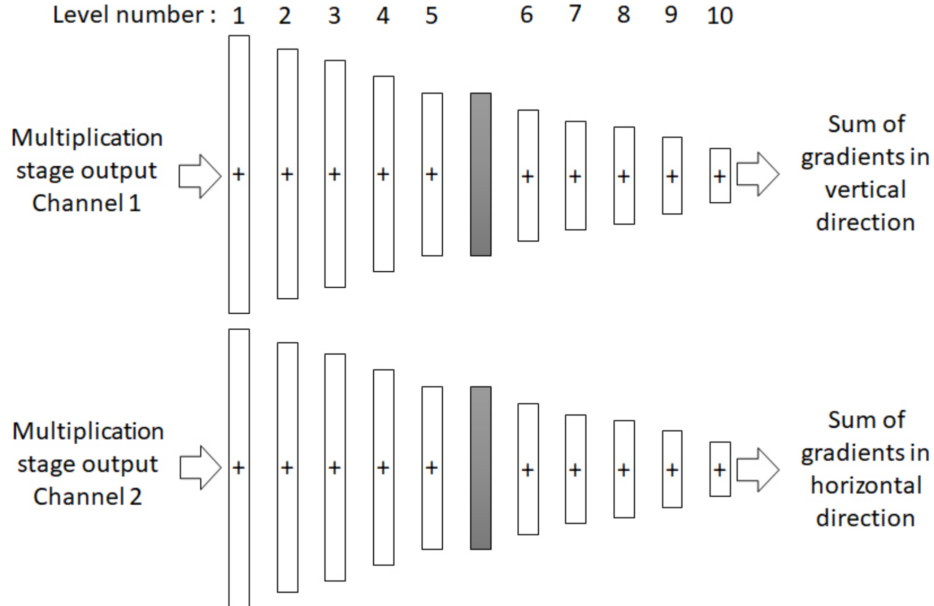

Figure 4.11: Two parallel 10-level adder trees for two channels. Each level has half of the number of adders in the previous level. The solid bar represents a pipeline register which is used to make the critical path shorter.

and L is the number of long pairs which is 870. Note that c_1 and c_2 are precomputed constants and we can use their values for long-pair calculations.

The first step of long-pair calculation, which is shown in Fig. 4.10, is to compute the difference between the intensity of pixels in each long pair. In this step, we compute 870 subtractions in parallel in one clock cycle which results in 9-bit outputs to accomodate overflow. The second step is multiplication by c_1 and c_2 coefficients. For this step, we should multiply the 870 differences once by c_1 and once by c_2. Therefore, we have two channels of multiplication with the total number of 1740 multiplication operations which we perform in parallel in one clock cycle. We use add and signed arithmetic shifts instead of complex multiplier circuits, since c_1 and c_2 are constants. In this way, fewer hardware resources are used for this step and the output is computed in a single clock cycle.

The third step is the summation step as shown in (4.6). For this step, we use two parallel 10-level adder trees. The input of the first level for each adder tree is the results of the multiplications of each channel. The two outputs of this step are the values of $g(x)$ and $g(y)$. The block diagram of this step is shown in Fig. 4.11. We use a pipeline register between levels 5 and 6 since this step is one of the critical timing paths of the design. The input values of the first level of adder tree, which are the output of the multiplication step, are 18-bit integers. The output of each level of the adder tree should have one more bit than the input level to accommodate overflow in addition. Increasing the number of bits in each level will result in 28-bit output at the final stage. In this step, we take advantage of the fact that in the next step, the result of the adder tree for the vertical direction is divided by the result of another adder tree for the horizontal direction. Therefore, if both these values are divided by a constant, the final result will not change.

In order to save hardware resources, we use a stepwise approximation in the adder tree. At each level, we divide the result of the addition by a specific power of two. Division by power of two is equivalent to

Table 4.5: Adder tree number of additions and approximations

Adder tree levels	Number of additions in each level	Pre-approximation output (bits)	Approximate output (bits)
1	864	19	16
2	432	20	15
3	216	21	14
4	108	22	13
5	54	23	12
6	27	24	12
7	14	25	12
8	7	26	12
9	4	27	12
10	2	28	12

selecting bits and propagating them to the next level. By using this method, the final result of the next step (after division) is comparable to the result using accurate addition using all bits. For the first level, we divide the output by 8. For the second to fifth level, we divide the output by 4. For the rest of the levels, we divide the output of each level by 2. We choose these values empirically to minimize hardware resource utilization while maintaining similar accuracy. Table 4.5 shows the number of bits before approximation and after approximation in each level. The fourth step of the long-pair calculation is orientation computation. In this step, we compute the multiplication of $g(x)$ and different values of $tan(\theta)$ limits and compare them to $g(y)$ as in (4.9):

$$g(x)tan(\theta_{i+1}) > g(y) \geq g(x)tan(\theta_i) \tag{4.9}$$

To make the descriptor rotation invariant, we should rotate the samples based on the orientation from step four. Based on the orientation value, we dynamically select the rotated samples using multiplexers working in parallel after the pipeline registers. This method does not require reading any data from memory and is faster than computing the rotated sample locations. The block diagram of the fifth step, which is short-pair computation, is shown in Fig. 4.12. After finding the correct angle, instead of rotating the patch, we use the sample pairs for that specific orientation. In this step, we have 1024 36-input multiplexers and 512 8-bit comparators. The orientation is used as a selector of the multiplexers. The output of the multiplexers are the rotated samples and the output of each two multiplexers corresponding to one short pair are connected to a comparator. Finally, the output of the 512 comparators are concatenated as a single 512-bit vector which is the final descriptor of a keypoint.

4.4.2 Multi-scale BRISK

We use two similar parallel pipelines to implement multi-scale description. This architecture is shown in Fig. 4.7. The only difference between the two pipelines is the streaming input data. The input data of the second pipeline is extracted from the second line-buffer. An 11×11 patch from the second line-buffer enters the filter unit of the second pipeline. Then, the filtered result enters the third line-buffer which produces a 17×17 patch as input to the second pipeline.

The first line-buffer has the dimensions of $W\times11$ (recall W is the image width). We choose this size

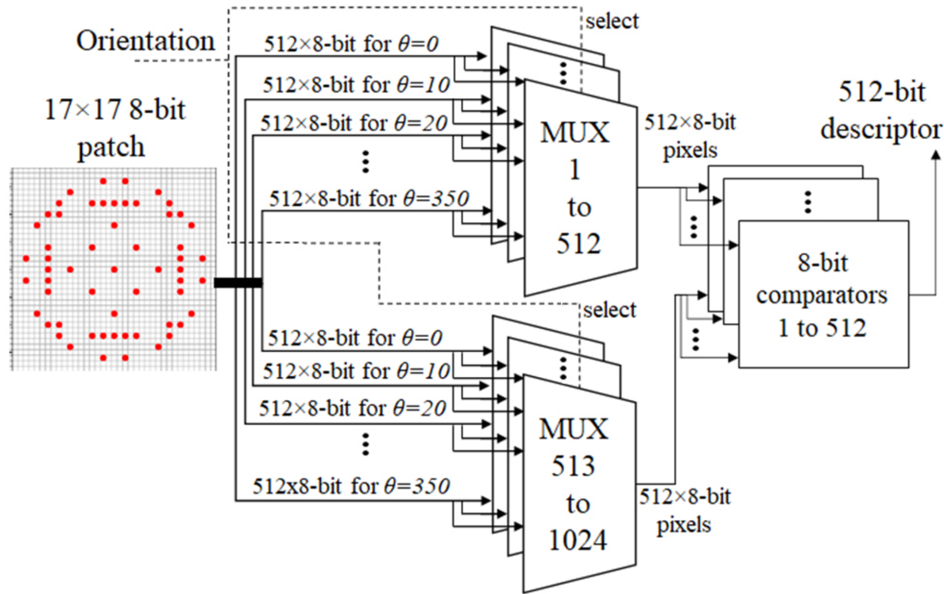

Figure 4.12: Short pair computation step. In this stage, we have 72 busses which connect the pixels in the sampling pattern to the multiplexers. Each multiplexer selects one group of samples and passes it to the 512 comparators. The descriptor comprises the outputs of the comparators.

since the purpose of this line-buffer is to provide parallel input for the filtering unit, which contains a weight window of 11×11. The second line-buffer has the dimensions of $W\times17$. This line-buffer has two outputs. The first one is a 17×17 patch which is the input of the first pipeline and the second output is an 11×11 patch which enters the second pipeline stage. The third line-buffer has the dimension of *half of the second line-buffer width*×17. Since the original input enters the system each clock cycle and we are scaling each line-buffer to have approximately half of the size of the previous one, valid data in the second line-buffer is available every two clock cycles. Similarly, the valid data in the third line-buffer is produced once every four clock cycles.

Our design provides the output of two scales for the modified BRISK algorithm. Additional scales can be added to the design by adding more pipelines similar to the second scale as shown in Fig. 4.7.

4.5 Results and discussion

In this section, first, we discuss the performance of our new sampling pattern. Second, we present the advantages of clock gating on our implementation and the benefits of sharing FPGA resources for the multiplication stage. Then, we present the FPGA resource usage of our implementation of the BRISK algorithm in detail. After that, we demonstrate the accuracy of our implementation.

4.5.1 Assessment of the new sampling pattern

It is important to note that we use even rows and columns in our implementation but if odd rows and columns are used, the concept and the result will be the same. Using only the samples in even rows and

Table 4.6: Number of required registers for pipeline implementation of the BRISK algorithm in original BRISK and our proposed pattern

Stages of the pipeline implementation	Original BRISK**	Our proposed pattern
Long-pair subtraction, multiplication, and short-pair computation	5×(33×33)	5×(17×17)
Long-pair summation and orientation	2×(33×33×2*)	2×(17×17×2*)
Total number of registers for passing through the pipeline	7×33×33=7623	7×17×17=2023

*We have two channels in the long-pair summation stage and in the input of long-pair orientation stage.

**Values of this column are our estimation of the resources for implementing the original BRISK algorithm.

even columns gives us an opportunity to process an image of diminished size without loss of information. Therefore, we can have a more efficient design regarding the resource usage on the FPGA. In the original BRISK, each patch has a minimum size of 33×33. If we load the patch and the surrounding pixels from the memory so that we can smooth the patch around the samples in the borders, we should use a 33×33 patch. Therefore, implementing the original BRISK for 8 scales requires $W \times 61.875$ registers for line-buffers according to (4.10). However, by using the new sampling pattern, we can reduce the number of required registers to about 27 times the width of the image as in (4.11). As shown in (4.10) and (4.11), for each scale, the width of the image is reduced to half size and therefore, the width of the line-buffer in each scale is equal to width of the image divided by the scale factor of that scale.

$$\begin{aligned} R_{BRISK} &= 33 \times (\frac{W}{1} + \frac{W}{2} + \frac{W}{4} + \frac{W}{8}) \\ &= 61.875 \times W \end{aligned} \tag{4.10}$$

$$\begin{aligned} R_N &= 17 \times (\frac{W}{2} + \frac{W}{4} + \frac{W}{8} + \frac{W}{16}) + 11 \times W \\ &= 26.9375 \times W \end{aligned} \tag{4.11}$$

In (4.10) and (4.11), W is the width of the image, R_{BRISK} is the number of registers required for a 4-scale implementation of the original BRISK pattern line-buffers and R_N is the number of registers required for our design. We use 17×17 patches for the scales and we have a buffer of $W \times 11$ for prefiltering the first patch. Equations (4.10) and (4.11) show that we have 56% reduction in the number of registers in the design. Another advantage of reducing the size of the patch is that we propagate fewer number of signals though the pipeline. Table 4.6 shows the number of registers used in each pipeline stage. In total, the number of registers decreases by 73% which is another benefit of our proposed sampling pattern.

Table 4.7: Power consumption improvement using clock gating

Power metrics		Ungated clock (watts)	Gated clock (watts)	Improvement
Static power		0.486	0.482	1%
Dynamic power	Clocks	0.733	0.372	49%
	Signals	0.102	0.013	87%
	Logic	0.255	0.190	25%
	I/O	0.003	0.003	0%
	Total	1.093	0.578	47%
Total Power		1.579	1.060	32%

4.5.2 Effect of clock gating

The first two steps of an image matching system are keypoint detection and keypoint description. For the first part, we should apply the keypoint detection algorithm on all pixels in the image to identify the keypoints. Unless a dense description of all pixels is required, the description part is applied to patches around a detected keypoint which is the common procedure used in most applications. Our proposed architecture can produce dense descriptors for all pixels in the image. However, we can use the output of the detection stage as a control signal to produce the description results, only for the keypoints. In this way, the part of the circuit which is dedicated for description does not consume dynamic power when there is no keypoint detected.

To implement this feature, we use a clock gating method. We concatenate the detection bit to the 8-bit pixels in the second and third line-buffers. If the pixel is a keypoint, the 9th bit is 1; otherwise it is 0. We use the detection bit as the control signal for the pipeline registers on the FPGA. When a keypoint pixel reaches the end of the line-buffers, the 9th bit, which indicates if the pixel is a keypoint or not, is used as an enable signal for the clock routing to all registers at that scale.

Since the computation of each pixel requires 9 clock cycles in the descriptor pipeline, the clock of the pipeline registers stays active for 9 consecutive clock cycles after each keypoint. Table 4.7 shows the effect of gating the clock for the descriptor on power consumption versus using the same clock for all units in the circuit. In this table, the pipeline registers are controlled by the common clock of the circuit for the ungated clock design. As shown in Table 4.7, the static power is not affected much by this design decision. However, the dynamic power has improved by 47%. Since the static power is about the same in both cases, the total power has improved by 32%.

4.5.3 Sharing the multiplication stage

In this work, we implement the BRISK descriptor for two scales of an image. Since we are using only even rows and columns of an image and the image pixels are read at each clock cycle from the memory, the data in the pipeline of the first scale is only valid every two clock cycles. For the second scale, the data is valid every four clock cycles. We synchronize the two pipelines so that the data of the second pipeline is produced in the unused cycles of the first pipeline. We take advantage of this fact by sharing the hardware resources for the multiplication stage since it is a hardware consuming part of the pipeline. The idea is to use the same hardware resources for both scales. We use a multiplexer before the multiplication stage and according to the clock signal, we select the values from one of the pipelines. Therefore, this unit can produce valid results

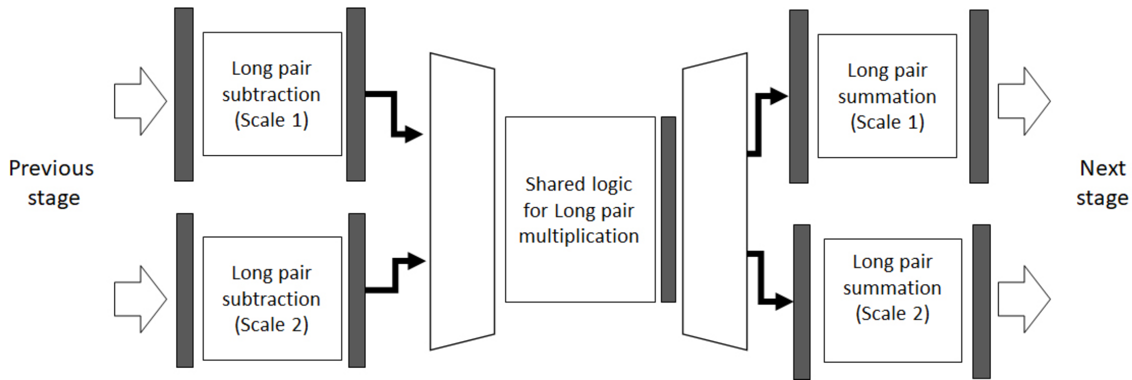

Figure 4.13: Shared multiplication logic between two scales. We use a multiplexer before the shared logic to select the data from one of the pipelines and a demultiplexer after the unit to propagate results to the next stages of the pipeline.

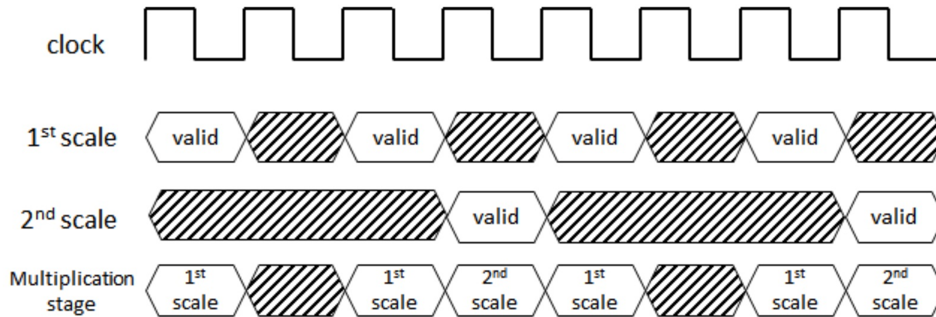

Figure 4.14: Timing of the valid data on the 1st and 2nd scales. The bottom waveform shows the allocation of the multiplication stage to each scale.

in three out of every four clock cycles without any conflict between the two pipelines. This will lead to a 14% less usage of LUT (Look Up Table) resources. Figure 4.13 shows the idea of sharing the multiplication stage between two scales. Figure 4.14 shows the timing of the valid data on each scale and the allocation of the multiplication stage.

4.5.4 Implementation results

We use the KCU105 FPGA board [109] which contains a Kintex® Ultrascale™ FPGA for all the experiments in this work. The resource consumption and timing information of the design are provided in this section. Table 4.8 shows the amount of resource usage of the proposed system with two scales on a 1920×1080 image resolution. We use the notations $\text{HWBRISK}_{initial}$ and HWBRISK_{final} to indicate the importance of adder tree bit-width approximation and sharing the multiplication stage between two pipeline scales. Note that the $\text{HWBRISK}_{initial}$ column shows the FPGA resource usage before sharing the multiplication stage and approximating the adder tree. The HWBRISK_{final} column shows the final results of our implementation. As shown in Table 4.8, our implementation does not use any DSP core or block RAMs of the FPGA and all the intermediate computing results are stored in the registers. This design choice reserves FPGA resources for other computation. Multiplications and divisions are implemented on LUTs. In addition, the locations

Table 4.8: Total resource usage for two scales of BRISK descriptor on KCU105 FPGA board

Resources	$\text{HWBRISK}_{initial}$		HWBRISK_{final}	
	Amount used	Percentage	Amount used	Percentage
LUT	182375	75%	147029	61%
LUTRAM	11400	10%	11400	10%
FF	124502	26%	189310	39%
Block RAM	0	0	0	0
DSP	0	0	0	0

of long pairs and short rotated pairs are stored in LUTs. The LUT usage is reduced by 14% after sharing the multiplication stage and approximating the adder tree in the long-pair summation stage. The trade-off for using less LUTs is an increase in flip-flop utilization. Since we use extra registers for storing the data in multiplication stage, there is an increase of 13% in flip-flop numbers. Table 4.9 illustrates the FPGA resource usage of each stage of the design.

Table 4.9: Detail of resource usage in various steps of the BRISK descriptor

Design modules	$\text{HWBRISK}_{initial}$			HWBRISK_{final}		
	LUT resources	LUT as Memory	LUT as Logic	LUT resources	LUT as Memory	LUT as Logic
First Line-buffer	5324	5280	44	5324	5280	44
Second Line-buffer	4148	4080	68	4148	4080	68
Third Line-buffer	2108	2040	68	2108	2040	68
Filter unit	2168	0	2168	2168	0	2168
Long-pair subtraction	6050	0	6050	6050	0	6050
Long-pair multiplication	36223	0	36223	47202**	0	47202
Long-pair summation*	16389×2	0	16389×2	14271×2	0	14271×2
Long-pair orientation	1341	0	1341	577	0	577
Short-pair comparison	6589	0	6589	6589	0	6589
Controller	375	0	375	375	0	375

*We have two channels of long-pair summation in each pipeline.

**This value is for two scales since it is shared between two pipelines.

As shown in Table 4.9, the long-pair summation stage consumes most of the LUT resources of the FPGA. The reason is that the outputs of the long-pair multiplication stage are two channels of 870 18-bit values which enter the summation stage. These 18-bit values are added together using 10-level adder trees. We can see that by approximating the adder tree in long-pair summation, the number of LUTs have decreased from 16389×2 in HWBRISK$_{initial}$ to 14271×2 in HWBRISK$_{final}$. The ×2 notation emphasizes that we are using two channels of summation in each pipeline. Although the number of LUTs for the long-pair multiplication stage in HWBRISK$_{final}$ (47202) is more than that for HWBRISK$_{initial}$ (36223) in Table 4.9, the total LUT resources for this unit is decreased (from 2×36223 to 47202). The value presented in the column HWBRISK$_{initial}$ is for the multiplication stage in one pipeline. On the other hand, the LUT resources shown in the HWBRISK$_{final}$ column is the amount used for the shared unit for two pipelines, which should be compared with the resource usage of two pipelines (72446). Therefore, we have reduced the LUT usage by 35% for this stage. The maximum frequency of the design is 168MHz which leads to 78 fps for the full HD image size that is 1920×1080 pixels.

Table 4.10: A summary of design metrics for recent FPGA implementations of binary descriptors

Design metrics	Soleimani et al. [34]	Fang et al. [105]	Sun et al. [33]	Huang et al. [110]	Ulusel et al. [108]	HWBRISK$_{final}$
Algorithm	AKAZE	ORB	ORB	BRIEF	BRISK	BRISK
FPGA	Kintex® Ultrascale™	Altera Stratix® V	Zynq® Ultrascale+™	Kintex-7®	Zynq® 7020	Kintex® Ultrascale™
LUT	184872	25648	28168	80472	25575	158429
BRAM	18Mb	9.44Mb	1.47Mb	35kb	396Kb	0
DSP	31	8	33	0	–	0
FF	65028	21791	9528	112166	7115	189310
Image resolution	1280×720	640×480	1920×1080	512×512	800×480	1920×1080
Frequency (MHz)	100	240	200	100	111	168
Frame rate (fps)	304	67	108	310	147	78
Throughput (pixels/cc)*	2.8	0.085	1.12	0.81	0.51	0.96
Latency (cc)**	36	1166	89	123	197	104
Total Power (mW)	1095	–	873	–	2400	1060

* Throughput is the number of pixels processed per clock cycle (cc).

** Latency is the estimated number of clock cycles for 100 pixels.

We report the implementation design metrics of recently published FPGA-based binary descriptors in Table 4.10. Since the algorithms have varying types and number of computations, we cannot directly compare them based on the reported numbers. As an example, the implementation of ORB by Sun et al. [33] leads to a 256-bit descriptor while BRISK produces a 512-bit descriptor. In addition, for orientation estimation they process pixels in a circular patch around the keypoint (approximately 800 pixels). For BRISK, we select different pairs from the same patch and process about 1740 pixels. This is why resources for BRISK implementation are higher than for ORB implementation. In [34], only the scale space generation is implemented. Huang et al. [110] implement the BRIEF algorithm which does not use an orientation estimation stage.

In Table 4.10, we compare the resource usage of our design with that of [108] which is an FPGA-based design of the BRISK algorithm. Ulusel et. al [108] analyze the implementations of BRIEF and BRISK algorithms on an embedded CPU, an FPGA, and a GPU. For the FPGA implementation, they implement the pipeline for FAST detection and BRISK description only for one scale. They also use 11 of the Block RAMs on the FPGA. In our work, we report the resource utilization for two scales, and we store the precomputed patterns on the LUTs of the FPGA. We do not use any of the Block RAM resources of the FPGA. As a result, the LUT resources reported in [108] are fewer than in our work. Ulusel et al. [108] do not describe the details of the orientation estimation step. Since orientation estimation is a significant component of our design and we have employed extensive parallelism in the computations of this stage, we cannot fairly compare our design with that of [108] in terms of resources. Since the focus of this work is on comparison of hardware design metrics such as power, runtime, resource utilization, and energy, they have not reported the accuracy result of their BRISK implementation.

We compare the speed metric of our design and other work in Table 4.11. For the same image size, our design achieves a higher frame rate and throughput and lower latency with respect to the other BRISK implementation. We measure the speed of the BRISK algorithm on the first image of the Boat set in the Oxford Affine Covariant Regions dataset [111] on a CPU implementation (Intel Core-i7, 1.3 GHz processor with 6GB of RAM) to compare with our FPGA implementation. For CPU implementation, we use a C code version of the BRISK algorithm based on OpenCV libraries. For 1920×1080 pixel images, the CPU implementation can achieve 3.5 fps while our FPGA implementation achieves 78 fps, respectively. It is important to note that a larger number of keypoints can lead to increased processing time in the CPU implementation while the timing of our design is independent of the number of keypoints. This comparison indicates that the speedup in our design ($HWBRISK_{final}$) is due to our implementation rather than the BRISK algorithm itself. We also provided the results of [105] to compare with our design. Our design achieves higher frame rate with lower frequency which leads to lower dynamic power.

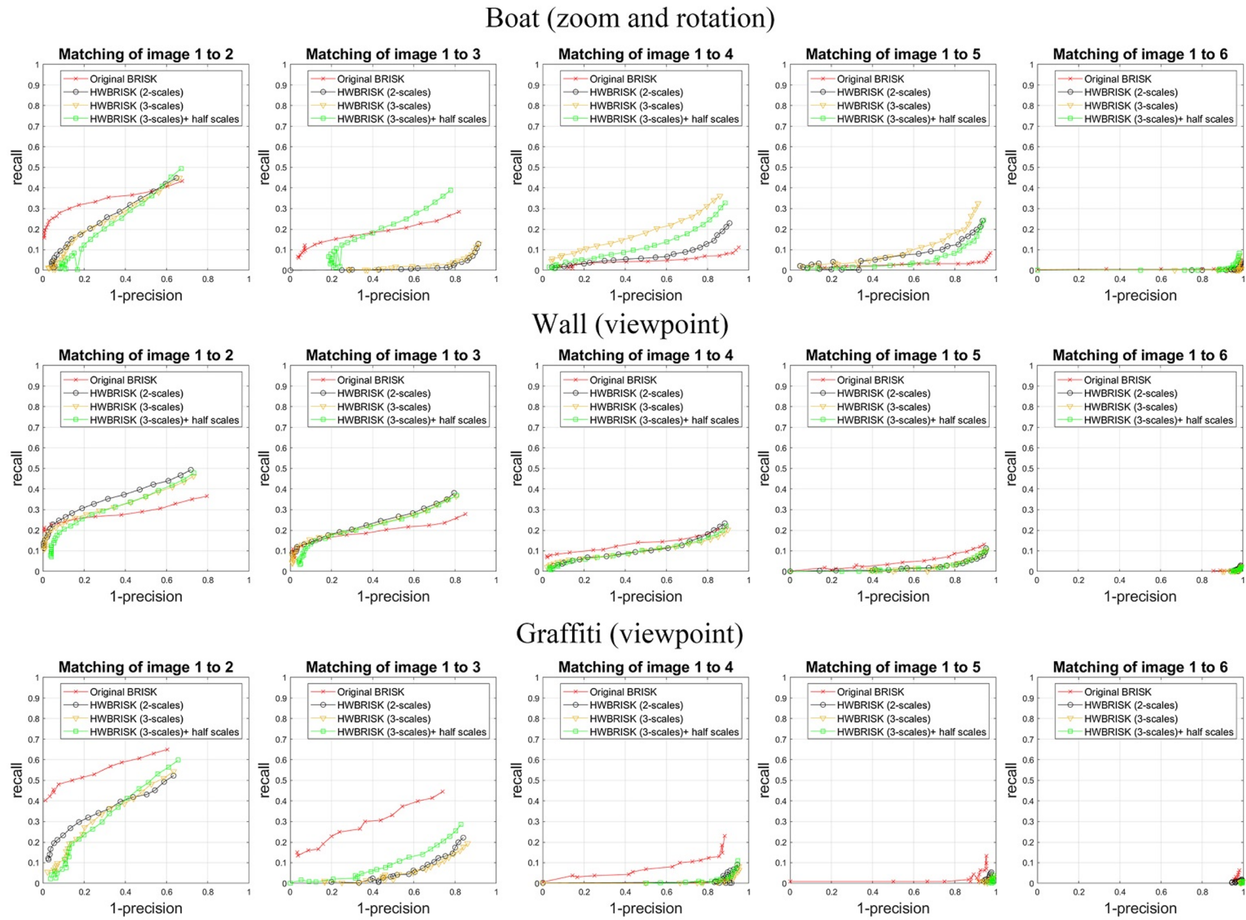

Figure 4.15: Matching results using recall vs. 1−precision curve for Boat, Wall, and Graffiti image sets of the Oxford Affine Covariant Regions dataset [111].

Table 4.11: Comparison of our design and other work in speed metric

Reference	Algorithm	Image size	Frequency	Frame rate
Fang et al. [105]	ORB	640×480	240 MHz	67 fps
Our design	BRISK	640×480	168 MHz	516 fps
Ulusel et al. [108]	BRISK	800×480	111 MHz	147 fps
CPU implementation*	BRISK	800×480	2.3 GHz	6.5 fps
Our design	BRISK	800×480	168 MHz	413 fps
CPU implementation*	BRISK	1920×1080	2.3 GHz	3.5 fps
Our design	BRISK	1920×1080	168 MHz	78 fps

*Intel Core-i7 processor with 6GB RAM in 1.3GHz.

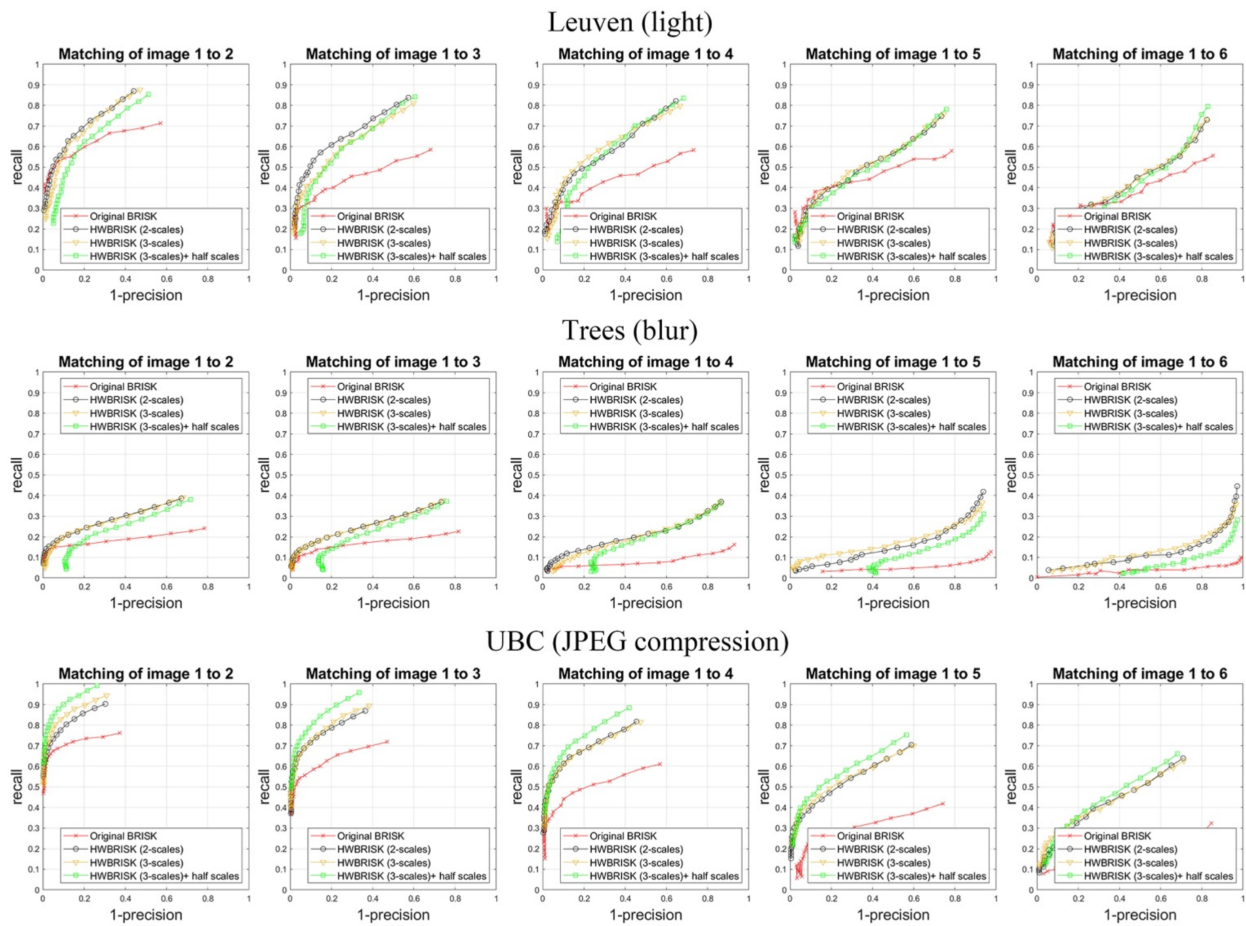

Figure 4.16: Matching results using recall vs. 1−precision curve for Leuven, Trees, and UBC image sets of the Oxford Affine Covariant Regions dataset [111].

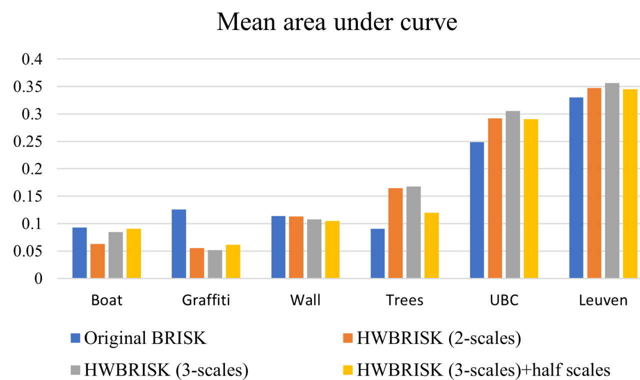

Figure 4.17: Comparison of the original BRISK algorithm and HWBRISK$_{final}$ in different scales using mean AUC (area under curve) for images from the Oxford Affine Covariant Regions dataset [111].

4.5.5 Accuracy evaluation

In order to evaluate the correctness of our design, we tested our implementation on the Oxford Affine Covariant Regions dataset [111]. The Oxford dataset consists of a variety of image sets which have different transformations of a scene. These transformations include changes in scale, rotation, blur, light, and JPEG compression. Recent work such as [33] and [108] has used a subset of this dataset to evaluate implementation as this dataset is representative of common image transformations.

To compare our design with the original BRISK algorithm, we use a recall vs. $1-$precision curve. This curve demonstrates a trade-off between recall and precision and is commonly used in descriptor evaluation literature. High precision relates to a low false positive rate which shows the accuracy of the algorithm, and high recall relates to a low false negative rate which shows the percentage of accepted matches over found matches. A large area under the curve indicates both high recall and high precision. Figures 4.15 and 4.16 present the recall over $1-$precision curves for matchings between the first (reference) and the other images of each set. In these figures, we provide the matching results of the original BRISK algorithm, our proposed hardware (HWBRISK$_{final}$) with 2 scales, HWBRISK$_{final}$ with 3 scales, and HWBRISK$_{final}$ with 3 scales and 3 subscales. Each main scale is multiplied by two and subscales are levels between two main scales. To have a fair comparison, we use the FAST detector for the original BRISK descriptor as well.

Figure 4.15 shows the matching results of the Boat, Wall, and Graffiti image sets which have zoom, rotation, and viewpoint transformations. Figure 4.16 illustrates the matching results on the Leuven, Trees, and UBC image sets which contain light, blur, and JPEG compression transformations, respectively. Figure 4.17 shows the mean area under curve (AUC) of a variety of image sets. Each value in Fig. 4.17 is the mean of the AUCs for the five images shown in Fig. 4.15 and Fig. 4.16. A higher value of AUC relates to a larger area under the recall versus $1-$precision curve, which shows better performance. For example, the AUC value of HWBRISK$_{final}$ with two scales is 0.29 on the UBC dataset which is supperior to the AUC of the original BRISK algorithm which is 0.25. HWBRISK$_{final}$ shows more recall in the same precision in comparison with the original BRISK in the Leuven, Trees, and UBC image sets. As an example for the matching of image 1 to 2 of the Leuven dataset in Fig. 4.16, HWBRISK$_{final}$ with two scales achieves a recall of 79% for 1-precision of 0.4 while the original BRISK attains 68% at the same precision. For the Boat and Wall image sets, the matching results are comparable. The original BRISK has better matching results in the Graffiti image set.

We use an exact fixed-point MATLAB® simulation model of our hardware design which generates identical output as the descriptor for this test. In all cases, we use the FAST algorithm as the detector since it is very similar to AGAST, which is the detector used in the original BRISK algorithm. The only difference between them is in the decision tree which modifies the order of pixel testing to increase the speed of the detector. The results show that our proposed design is comparable in accuracy to the original BRISK algorithm. In most of the sets including Leuven, Trees, UBC, and Wall, our model achieves higher recall in the same $1-$precision value while in the Graffiti image set, the original BRISK performs better. Table 4.12 shows the direct comparison of mean AUC on Oxford imagesets for various sampling patterns. In this test, we use the same detector, filter method, and scale levels, with the only difference being the sampling pattern. HWBRISK$_{final}$ pattern achieves the highest AUC in comparison with other patterns shown in Table 4.12, and is comparable with the original BRISK. HWBRISK$_{final}$ achieves higher AUCs as shown in Fig. 4.17 since the parameters of our complete design are tailored for the proposed sampling pattern.

Table 4.12: Direct comparison of the sampling patterns

Sampling pattern	Threshold T	Angle step n	mean AUC
Original BRISK (Fig. 4.1)	–	–	0.14
HWBRISK$_{final}$ (Fig. 4.5 (b))	1.00	36	0.12
Pattern 1 (Fig. 4.5 (a))	0.98	18	0.02
Pattern 2 (Fig. 4.5 (c))	1.02	18	0.10
Pattern 3 (Fig. 4.6 (a))	1.00	24	0.09
Pattern 4 (Fig. 4.6 (c))	1.00	48	0.11

4.5.6 Discussion

In this section, we present a discussion on the characteristics of the proposed design for implementation of the BRISK algorithm on an FPGA. The main goal in this design is to achieve higher speed. Therefore, we designed each part of the algorithm to operate in parallel. For example, all the subtractions in long-pair calculations are computed in parallel. This design decision leads to higher resource usage which is shown in Table 4.9. There are two solutions to reduce resource usage as a trade-off for speed. First, we can use shared hardware resources for each stage. As an example, we can use fewer number of subtraction modules or multipliers and perform the computations with more latency. Second, we can approximate the data between stages and reduce the bit-width of the data-path. As shown in Table 4.9, long-pair summation consumes FPGA resources more than other parts of the system. The reason for this high resource consumption is that we implement two 10-level adder trees for which the inputs of the first level are each 18-bit. If resource consumption is critical in a specific application, we can decrease the bit-width of long-pair calculation units and approximate the computations.

Precision-recall curves shown in Fig. 4.15 and Fig. 4.16 demonstrate that HWBRISK$_{final}$ has higher accuracy than the original BRISK algorithm under image variations such as light, blur, and JPEG compression. It also has higher accuracy in most of the images in the Boat dataset which has variations in scale and rotation. However, the original BRISK performs better on the Graffiti dataset possibly due to the complexity of the Graffiti images and the more uniformly-distributed pattern of the original BRISK algorithm which handles orientation estimation more preciesly.

The proposed design can be used as a part of a larger vision processing system. The input image can be read from memory or received directly from a camera. Our design can be added to various keypoint detectors for various applications. The keypoint detector part can be implemented in hardware or software and it does not affect the efficiency of the descriptor part. In addition, we can store the descriptors in on-chip memory, off-chip memory, or output them using a streaming protocol depending on the application. Although we focused on the BRISK algorithm, the idea of using a hardware-aware sampling pattern that facilitates hardware implementation could be adapted to other binary descriptors as well.

4.6 Conclusion

In this work, we introduced a multi-scale FPGA-based implementation of the BRISK descriptor. We presented a new sampling pattern for the BRISK algorithm with a similar number of sampling points, which reduced the number of registers for line-buffers by more than 50% and the pipeline registers up to 73%. The

proposed design is fully pipelined and achieves a maximum operating frequency of 168MHz. For images with 1920×1080 resolution, our proposed implementation has a frame rate of 78 fps.

There are multiple potential enhancements that can be addressed in the future of this research. First, we can implement a gated filter unit after the keypoint detection as the next stage of the pipeline to reduce power consumption. Second, we can replace the FAST detector with a more complex detector. Since keypoint detection is an early stage of an image matching system, choosing an appropriate detector can improve the accuracy depending on the application. Finally, since the image may be already loaded on on-chip memory in many applications, designing a non-streaming input architecture for the BRISK algorithm is a potential future path for this research.

Chapter 5

Learned Fusion of Complementary Local Descriptors for Improved Image Matching Accuracy

Local descriptor algorithms are foundational in computer vision applications such as image matching and vision based measurement. Some local descriptor algorithms extract features containing similar information from images while others extract complementary information. In this work, we investigate the advantages of fusing a binary and a non-binary local descriptor algorithm. We propose and compare three methods to combine descriptor algorithms. The first method combines the non-binary and binary descriptors using a weighted summation of their individual descriptor distances with learned weights. Our second method converts the non-binary descriptor into a binary descriptor by learning a threshold and concatenates the converted binary vector with the other binary descriptor. The third method scales the binary descriptor vector and concatenates it with the non-binary descriptor. Parameters for fusing the descriptors are learned for each of the three methods and the methods are evaluated on the HPatches, Brown, and Oxford datasets using various evaluation measurement metrics. Our proposed methods are generic, have minimal dataset dependency, and are computationally efficient in comparison with deep neural network descriptors. Our experimental case study shows that our methods can improve mean Average Precision (mAP) as demonstrated with cross-validation using the HPatches dataset.

5.1 Introduction

Image matching is an important technique in computer vision with many applications including visual measurement [112], structure from motion [113], visual simultaneous localization and mapping [114], 3D reconstruction [115], and object recognition [116]. Image matching involves finding a transformation from one image to another image of the same scene or object of interest. The matching process comprises several sequential steps that include scale-space generation, keypoint detection, patch description, keypoint matching, and outlier removal.

In this work, we focus on patch description. The algorithms used in the description stage are known as

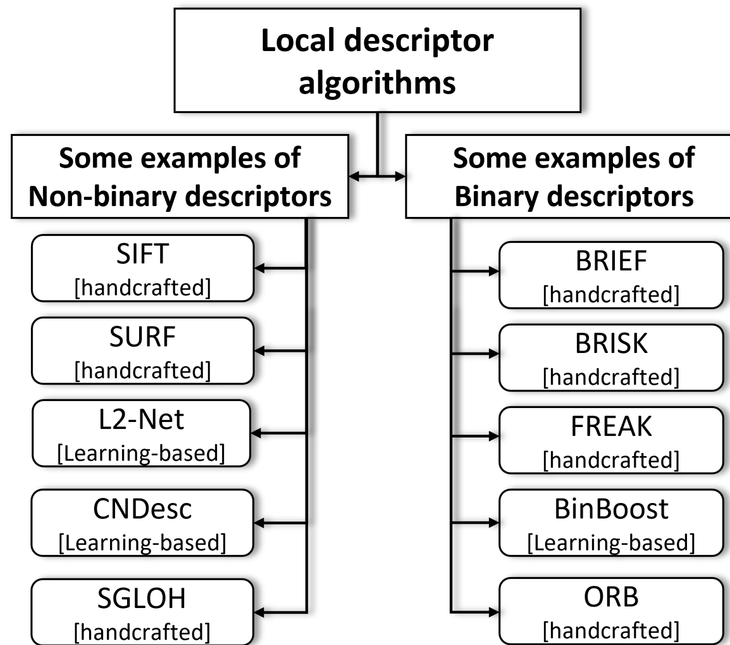

Figure 5.1: Categorization of local descriptor algorithms into non-binary and binary, and handcrafted and learning-based algorithms.

local descriptor algorithms. Local descriptor algorithms are a basis for many Vision Based Measurement (VBM) [117] applications such as bubble velocity measurement [118] and depth measurements [119]. Local descriptor algorithms can be categorized into *non-binary* and *binary* based on the generated output vector as shown in Fig. 5.1. Non-binary local descriptor algorithms such as Scale Invariant Feature Transform (SIFT) [19], Speeded Up Robust Features (SURF) [16], L2-Net [18], CNDesc [120], and Shifted Gradient Location and Orientation Histogram (SGLOH) [17] typically lead to higher precision as they usually extract and process more information from the image patches.

In contrast, binary local descriptor algorithms such as Binary Robust Independent Elementary Features (BRIEF) [20], Binary Robust Invariant Scalable Keypoints (BRISK) [22], Fast Retina Keypoint (FREAK) [21], BinBoost [27], and Oriented FAST and rotated BRIEF (ORB) [12] are introduced to increase the speed of the local descriptor algorithms. In another categorization by Ma et al. [4], some of the local descriptor algorithms such as SIFT, SURF, and BRISK are known as *handcrafted algorithms* as they are not tuned specifically for a training dataset. Other local descriptor algorithms such as L2-Net are known as *learning-based algorithms* as they learn the parameters of a model based on a training set. Learning-based algorithms such as L2-Net, D2-Net [121], Superpoint [122], ALIKE [123], and the work by Fan et al. [5] which are based on deep neural networks, are known to achieve higher accuracy with respect to handcrafted algorithms. However, as shown by Efe et al. [124] and Jin et al. [125], the performance gap between handcrafted and learning-based methods after tuning specific parameters for evaluation is not significant. In addition, learning-based descriptors which use neural networks require a higher number of operations for calculating the descriptor vectors with respect to handcrafted methods. In this work, we employ handcrafted descriptor algorithms as base models which are more computationally efficient than deep learning based descriptors.

Table 5.1: Summary of the three proposed methods

Method	**Description**
Weighted-fusion	Combining the computed matching distances of a binary and a non-binary descriptor algorithms.
Binary-fusion	Combining the descriptor vectors of the binary descriptor and a binary version of the non-binary descriptor algorithm.
Non-binary-fusion	Combining the descriptor vectors of a non-binary version of the binary and the non-binary algorithm.

Another advantage of handcrafted methods in comparison with deep learning based methods is better suitability for implementation on industrial instruments and hardware platforms such as Field Programmable Gate Arrays (FPGAs). Many FPGA-based implementations of local descriptors such as [40], [126] have been proposed in recent years to increase the speed of practical image matching applications. Parallel computation, reconfigurability and low power consumption are the main advantages of FPGA-based implementations.

In this work, our proposed methods are designed to be suitable for parallel processing so that the fusion requires minimal additional latency. We focus on combining methods for handcrafted local descriptor algorithms since the hardware implementation of handcrafted local descriptor algorithms results in higher energy efficiency and less amount of computation compared to hardware implementation of the learning-based algorithms [127].

We propose semi-learning-based methods which combine handcrafted descriptors by learning the parameters for fusion. In this way, we demonstrate an increase in the accuracy of handcrafted algorithms but with a lower number of operations than common learning-based algorithms.

Using different techniques for generating the descriptor vectors in binary and non-binary local descriptor algorithms leads to having complementary information embedded in their descriptor vectors. Our proposed methods in this work are generic and applicable to any non-binary and binary local descriptor algorithms. To demonstrate the computational performance and improved accuracy, we aggregate the information extracted using a popular non-binary (SIFT) and a commonly-used binary (BRISK) local descriptor algorithm as a case study. The reason that we choose these two algorithms is their popularity as baseline descriptor algorithms and due to their relatively higher performance in accuracy in comparison with other local descriptor algorithms [23], [128]. However, the methods proposed in this work are not limited to SIFT and BRISK algorithms and can be used to combine any other local descriptor algorithms as well. We use the terminology given in Table 5.1 to facilitate reference to our methods.

The remainder of this paper is organized as follows. In section 5.2, we briefly review the SIFT and BRISK algorithms and prior work which combines local descriptor algorithms to improve accuracy. In section 5.3, we describe our three proposed methods in detail. In section 5.4, our experimental results are presented and finally, we conclude this work in section 5.5.

5.2 Related work

In this section, first, we discuss the work that use combination of descriptors to achieve higher accuracy. Then, we briefly review BRISK and SIFT algorithms to illustrate their complementary nature.

5.2.1 Combining descriptors

Combining descriptor algorithms has been previously proposed in the literature [28], [129]–[131]. As an example, Gao et al. [28] combine four binary local descriptor algorithms, Receptive Fields Descriptor (RFD) with rectangular pooling area (RFD_R), RFD with Gaussian pooling area (RFD_G) [132], BinBoost [27], and Boosted Gradient Maps (BGM) [133], by using a weighted summation of their distance. They use a rank-SVM algorithm to learn the weights for fusing the descriptors. For evaluation, at 95% recall they achieve a lower false positive rate in comparison with most other binary and non-binary descriptors.

Salameh et al. [129] use a combination of SIFT, SURF and ORB algorithms for loop closure in visual SLAM applications. They use the set of keypoint descriptor vectors as input to a bag-of-words model. In the next stage, the descriptor vectors are used to construct Bayesian filter models and ensemble learning algorithm for loop closure detection. They show that using a fusion of local descriptor algorithms leads to a higher recall value than using the single descriptors individually. However, the bag-of-words technique to combine the local descriptor algorithms requires high similarity measurement between the training and testing sets which reduces the generality of their proposed model.

Husain et al. [134] use a fusion of binary local descriptor algorithms for large scale image retrieval in mobile scenarios. They propose Binary Robust Visual Descriptor (B-RVD) which combines BRISK, FREAK [21] and BRIGHT [135] for description while using the keypoint detection technique proposed by the BRISK algorithm. The proposed pipeline in [134] reports improvement in mean Average Precision (mAP) measurement in large scale image retrieval.

Fan et al. [131] propose a combination of BRIEF [20] and FREAK [21] binary descriptor algorithms by computing the summation of weighted binary descriptor vectors. They learn the weights for each element of the descriptor vectors to show the importance of each of the comparison bits. Although this method improves the accuracy with respect to the two baseline binary descriptors, it is highly dependent on the learned dataset as the importance of the position of the sample points and comparisons may differ from one dataset to another. Adding weights to individual elements of a binary descriptor reduces the accuracy of orientation estimation specially for the descriptors with uniformly distributed sampling pattern such as FREAK.

Yang et al. [136] propose a deep neural network method for description of image patches by using complementary data in training. They add an additional network stream to the initial network architecture which is trained on the data that is not handled properly by that architecture. By optimizing the two networks at the same time, their network streams learn to extract complementary data which improves the overall accuracy. Although deep learning methods such as [136] achieve better results in terms of accuracy, they are not as computationally efficient as handcrafted and non-deep learning based descriptors.

Liu et al. [137] propose a new binary descriptor, Features Combined Binary Descriptor based on Voted Ring-sampling pattern (BDVRP), which combines the intensity information of the pixels with gradient features. They show the effect of using complementary features for attaining higher accuracy using their proposed handcrafted descriptor algorithm. However, BRVRP is not extendable to combine features of the

other algorithms. Our proposed fusion methods can be applied for combination of any non-binary and binary descriptor algorithms.

5.2.2 A brief review of the SIFT and BRISK algorithms

The SIFT algorithm is introduced by Lowe [19]. After finding the keypoints, a 16×16 patch around each keypoint is divided into 16 blocks of 4×4 pixels. An 8-bin histogram of gradients is created in each block. The SIFT descriptor vector, which is a vector of 128 floating-point elements, is generated by concatenating the features from all blocks. SIFT is popular due to its higher accuracy in comparison with other handcrafted local descriptor algorithms. The disadvantage of the SIFT algorithm is its high number of computations which can result in a lower speed than other local descriptor algorithms. However, there are many variations proposed for improving SIFT, for example root SIFT [138].

The BRISK algorithm is proposed by Leutenegger et al. [22]. The BRISK local descriptor algorithm is based on a binary comparison test which was originally proposed by Calonder et al. [20]. BRISK has more robustness to rotation changes than the BRIEF algorithm. From the set of all pairs of pixels in the symmetrical sampling pattern, 512 shortest distances (short pairs) are used for patch description. BRISK computes an orientation vector for each patch by computing the angle between the summation of all the gradients in horizontal and vertical directions for all 876 long pairs.

After computing the orientation of the patch, the sampling pattern is rotated by that orientation so that the descriptor becomes rotation invariant for all patches. Then, the short pairs are used to perform the binary comparison test and the final BRISK descriptor vector is constructed by the concatenation of the binary values. The high number of operations required for the orientation estimation and compensation steps of the BRISK algorithm has resulted in lower speed of execution with respect to other binary descriptors such as ORB. However, the BRISK local descriptor algorithm has attained higher accuracy in comparison to other binary local descriptor algorithms [23], [128].

Fig. 5.2 demonstrates one of the differences in using SIFT and BRISK algorithms for describing a patch. Each 8 consecutive elements of the SIFT descriptor vector are extracted locally from one of the 16 neighborhoods of pixels in the image patch. In BRISK however, the short pairs used for description can be in various directions and are not limited to locally separated neighborhoods. In addition, BRISK uses the direct comparison of the smoothed pixel values to generate descriptors while SIFT creates histograms of orientations which of course vary with pixel values. The differences among the description techniques of BRISK and SIFT result in complementary information extracted by these two local descriptor algorithms which can be combined to attain higher accuracy.

In this paper, we investigate various configurations for combining a non-binary and a binary local descriptor algorithm to benefit from the information extracted by more than one descriptor. The first method (weighted-fusion) expands the idea proposed by Gao et al. [28] by combining a binary and a non-binary descriptor using weighted distance values. In the second method (binary-fusion), the non-binary descriptor is transformed to the binary domain and then is concatenated with the binary descriptor. In the third method (non-binary-fusion), we keep the non-binary descriptor unchanged and learn a parameter to convert the binary descriptor so that it can be concatenated with the non-binary descriptor. In this way, we analyze the results of combining two types of local descriptor algorithms in each domain.

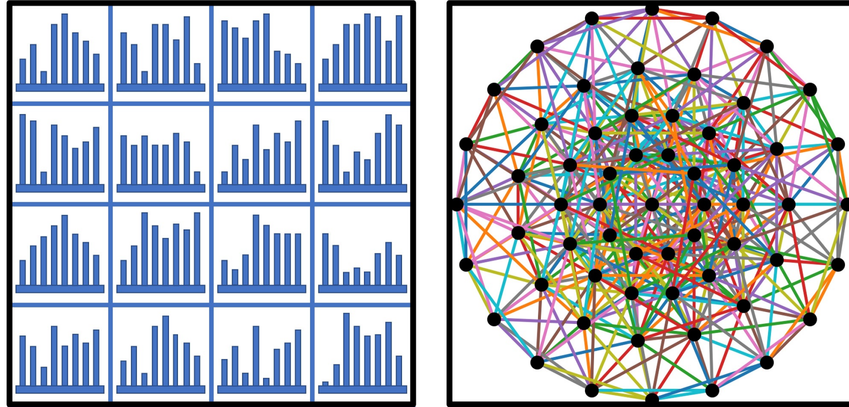

Figure 5.2: Comparison between descriptor vector generation of SIFT and BRISK. The left image shows an example of histograms generated by SIFT. Each histogram is extracted from the neighboring pixels in one of the 16 blocks in an image patch. The right image shows the location of 512 BRISK sampling pairs for the binary comparison test in an image patch.

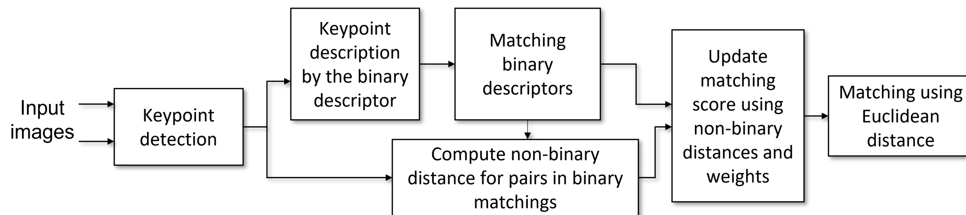

Figure 5.3: Block diagram of weighted-fusion for fusion of a non-binary and a binary descriptor algorithms using distance metric.

5.3 Fusing binary and non-binary descriptors

In this section, we describe our three proposed methods for the fusion of a binary and a non-binary local descriptor algorithms. This fusion can embed more complementary information in the descriptor.

5.3.1 Weighted-fusion

Weighted-fusion is based on the fusion of the distance values using a set of learned weights. In this method, we combine two local descriptor algorithms by the distances produced by each descriptor for each pair of keypoints. The overall block diagram and flow of weighted-fusion are presented in Fig. 5.3.

After the detection of keypoints, the binary local descriptor algorithm is used to extract features and generate the binary string for each image patch around the keypoints. After that, we find the matching pairs among the two images. In the matching process, the Hamming distance is computed among the binary descriptor vectors from the first image and the second image as shown in (5.1):

$$dist_{Hamming}(d_1, d_2) = \sum_{i=1}^{n} XOR\Big(d_1(i), d_2(i)\Big) \tag{5.1}$$

where d_1 and d_2 are the binary descriptor vectors from the first and second image, respectively, and $dist_{Hamming}$ is the Hamming distance. In (5.1), n is the number of elements in the binary descriptor vector.

Next, we compute the non-binary descriptor vector from the patches around the detected keypoints. Then, we measure the Euclidean distance of the non-binary descriptor vectors for each of the matched pairs that were originally produced by the binary local descriptor algorithm as shown in (5.2):

$$dist_{Euclidean}(d_1, d_2) = \sqrt{\sum_{i=1}^{n} (d_1(i) - d_2(i))^2} \tag{5.2}$$

where d_1 and d_2 are the descriptor vectors from the first and second images, respectively, and n is the number of elements in the descriptor vector and $dist_{Euclidean}$ is the Euclidean distance measurement.

In the next step, we use a weighted summation of the distances of the binary and non-binary descriptors to generate the overall distance value for each point. The overall distance value equation is shown in (5.3):

$$dist_{m1} = w_1 \times dist_{Euclidean} + w_2 \times dist_{Hamming} + b \tag{5.3}$$

where $dist_{m1}$ is the proposed new distance metric, $dist_{Euclidean}$ is the Euclidean distance measured using the non-binary descriptor vectors, $dist_{Hamming}$ is the Hamming distance measured using the binary descriptor vectors, and w_1, w_2, and b are the weights and bias terms, respectively. The weights are learned using stochastic gradient descent on a training dataset.

5.3.2 Binary-fusion

In binary-fusion, we directly combine the binary and non-binary descriptors after descriptor vector generation. Our goal is to concatenate the non-binary and binary descriptor vectors of each patch as a single binary vector. In addition to embedding complementary information in the descriptor vector, the binary-fusion method has advantages such as less memory footprint and a simpler distance measurement (Hamming distance) in comparison with non-binary descriptors. We use a previously learned threshold value to binarize the non-binary descriptor. If an element of the non-binary descriptor vector is greater than the threshold, its value in the new descriptor vector becomes 1; otherwise, it becomes 0. The proposed method for learning the value of the threshold is discussed in section 5.4. The conversion from a floating-point non-binary descriptor to a binary descriptor is shown in (5.4):

$$descriptor_{binary}(i) = \begin{cases} 1 \text{ if } descriptor_{float}(i) > \text{threshold} \\ 0 \text{ if } descriptor_{float}(i) \leq \text{threshold} \end{cases} \tag{5.4}$$

where i is in the range of 1 to n_{nb} (number of non-binary descriptor vector elements), $descriptor_{float}$ is the original non-binary descriptor vector, $descriptor_{binary}$ is the new binary descriptor vector, and threshold is

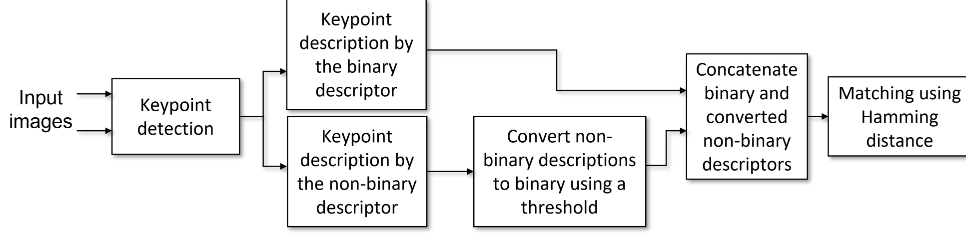

Figure 5.4: Block diagram of binary-fusion for fusion of a binary and a converted non-binary descriptors.

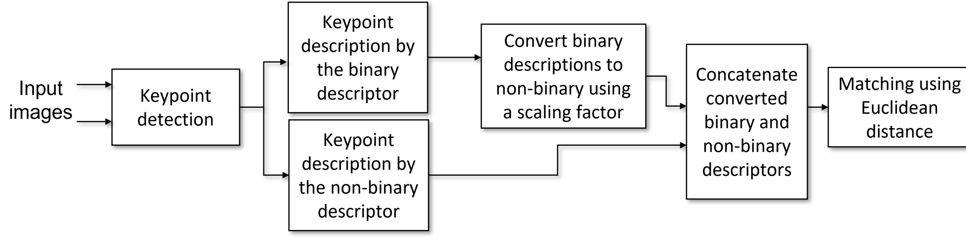

Figure 5.5: Block diagram of non-binary-fusion for fusion of a converted binary and a non-binary descriptors.

a learned parameter on the training set. The final descriptor is shown in (5.5):

$$d_{m2} = \{d_{B_1}, ...d_{B_{n_b}}, d_{CNB_1}, ...d_{CNB_{n_{nb}}}\} \tag{5.5}$$

where d_{B_i} and d_{CNB_j} (with i=1 to n_b, j=1 to n_{nb}) are the elements of the binary and converted non-binary descriptor vectors and d_{m2} is the final combined descriptor vector of this method, n_{nb} is the number of elements in the non-binary vector and n_b is the number of elements of the binary vector. Fig. 5.4 illustrates the overall block diagram of the binary-fusion method.

5.3.3 Non-binary-fusion

After considering the binary-fusion of two descriptors, we investigate the non-binary-fusion of the same descriptors for comparison in this section. In non-binary-fusion, we concatenate the binary and non-binary descriptor vectors in such a way that the final output is a floating-point non-binary descriptor. To do that, we multiply the binary descriptor vector by a learned scaling factor. Then, the scaled binary descriptor vector is concatenated with the non-binary descriptor vector. Fig. 5.5 shows the block diagram of non-binary-fusion. Equation (5.6) shows the concatenation of the non-binary and converted binary values:

$$d_{m3} = \{d_{NB_1}, ...d_{NB_{n_{nb}}}, W \times d_{CB_1}, ...W \times d_{CB_{n_b}}\} \tag{5.6}$$

where d_{NB_i} and d_{CB_j} (with i=1 to n_{nb}, j=1 to n_b) are the ith and j^{th} elements of the non-binary and converted binary descriptor vectors, respectively. W is a scaling factor and d_{m3} is the new descriptor vector generated using our proposed non-binary-fusion. The proposed method for learning the value of W is discussed in section 5.4.

Since the descriptor type is non-binary, we use Euclidean distance to measure the distances between the descriptor vectors from the first image to the second image.

By using d_{m3} from (5.6) in (5.2), we have:

$$dist_{Euclidean}(d_1, d_2) = \Bigg(\sum_{i=1}^{n_1} (d_{1_{NB}}(i) - d_{2_{NB}}(i))^2 + \sum_{i=n_1+1}^{n_2} (W \times d_{1_B}(i) - W \times d_{2_B}(i))^2\Bigg)^{1/2} \tag{5.7}$$

where $d_{1_{NB}}(i)$ and $d_{2_{NB}}(i)$ are the elements of the non-binary descriptor vectors from the first and second images, respectively, $d_{1_B}(i)$ and $d_{2_B}(i)$ are the elements of the binary descriptor vectors from the first and second images, n_1 is the number of elements in the non-binary descriptor vector and n_2 is the total number of elements in the d_{m3} descriptor vector. Equation (5.7) can be rewritten as (5.8):

$$dist_{Euclidean}(d_1, d_2) = \Bigg(\sum_{i=1}^{n_1} (d_{1_{NB}}(i) - d_{2_{NB}}(i))^2 + W^2 \times \sum_{i=n_1+1}^{n_2} (d_{1_B}(i) - d_{2_B}(i))^2\Bigg)^{1/2} \tag{5.8}$$

Equation (5.8) shows that W^2 is a weight for the binary part of d_{m3} and represents the importance of the binary descriptor vector in this equation. If $W = 0$, the effect of the binary descriptor vector for each patch becomes 0 and d_{m3} will be the same as the distance computed on the original non-binary descriptor vectors.

5.4 Experiments and results

As discussed in section 5.1, SIFT and BRISK algorithms are selected as our case study. The SIFT algorithm uses local histogram of orientations to generate the descriptor vector while BRISK uses intensity comparison in local patches. Since the methods and the information they extract from the image are different, using a fusion of their information can provide complementary data for better description of the image patches. For consistency of comparison, we use the BRISK detector algorithm for all reported experiments in this work due to its well known performance with respect to other detector algorithms [101]. We refer to weighted-fusion, binary-fusion, and non-binary fusion of SIFT and BRISK in this section as BRISK-SIFT, BRISK-bSIFT, and nbBRISK-SIFT, respectively.

In this work, we use three evaluation benchmarks based on the HPatches dataset [11], the Oxford Affine Covariant Regions dataset [111], and the Brown (Photo tourism) dataset [139] to evaluate our proposed methods. The HPatches dataset contains 116 sequences of 6 images with known homography. The Brown dataset comprises 64×64 image patches extracted from three sets of images, named Liberty, Yosemite, and Notre Dame. All image patches are normalized in terms of orientation and scale. Therefore, only the description stage of the algorithms is evaluated on this dataset. The Brown benchmark provides 500k sets of matching pairs and non-matching pairs for training. The Oxford dataset [111] contains 8 image sets each

having 6 images. The image sets present image transformations such as rotation, illumination, scaling, blur, and JPEG compression. The homography matrix for each of the transformations from one image to another is included which is used for evaluation. The advantage of evaluation on the Oxford image sets is that all stages of the description including scale-space generation, keypoint detection, orientation assignment, and patch description are evaluated.

5.4.1 Learning approaches

Several learning approaches for the proposed methods are used. For BRISK-SIFT, we use the training dataset to learn the weights using stochastic gradient descent. We first find the correct matches among the images of each sequence and extract the true positive pairs and false positive pairs using both BRISK and SIFT descriptor vectors. After that, we compute the distances of the same pairs and generate a set of true positive distances and a set of false positive distances for each pair of descriptors to be used as positive and negative samples for training. Then, we use stochastic gradient descent to find the best weight values to minimize the loss function as shown in (5.9).

$$loss = \sum_{i=1}^{n} max(dist_{m1}^{P}(i) - dist_{m1}^{N}(i) + c, 0) \tag{5.9}$$

where n is the number of distance pairs in the training batch, $dist_{m1}^{P}(i)$ and $dist_{m1}^{N}(i)$ are the distances calculated as shown in (5.3) for the ith true positive pair and false positive pair from the training dataset, respectively. Minimizing the loss function in (5.9) results in weight parameters (w_1, w_2, and b in (5.3)) that lead to a greater distance value for pairs of descriptors that are not a match and a lower distance value for pairs of descriptors that are a match in the training dataset. The weight parameters used in $dist_{m1}^{P}$ and $dist_{m1}^{N}$ do not change while computing the loss function and are updated after each iteration of the algorithm.

In order to learn the best threshold value for BRISK-bSIFT, we sweep the threshold value from 0 to 70 in steps of 3 and measure the mAP on the training set. Then, we select the threshold which results in the highest mAP on the training set.

For nbBRISK-SIFT, we learn a scaling factor to scale the BRISK descriptors and generate non-binary descriptor vectors from BRISK. We sweep the scaling factor from 0 to 250 in steps of 5 and select the highest mAP measurement on the training set. After that, we concatenate the SIFT and BRISK descriptor vectors and compute the Euclidean distance between the keypoints from the first image to the second image.

A fusion of descriptors requires more computation than that of a single descriptor. However, if the descriptor computations are run in parallel, on an FPGA platform for example, the added latency is minimal. Our focus in this work is on improving the accuracy of image matching applications.

5.4.2 Experimental results and measurements on HPatches dataset

In this section, we use the HPatches image sequence evaluation benchmark which contains 116 sets of 6 images. We use mAP which is a commonly-used measurement for image matching evaluation [11]. The mAP is measured as the area under the precision-recall curve. We use 5-fold cross validation on the whole dataset. In each fold, 80% of the sequences are used for training and 20% are used for testing. We use the training set to fine tune the parameters in each method. In each set of the HPatches sequences, we use the

first image as a reference image and the other 5 images as the target images for image matching. We use the following procedure for evaluating the matching process. After detecting the points in both images and matching them, the correct location of the points from the first image in the second image is calculated using equation (5.10):

$$\begin{bmatrix} x_{1,proj} \\ y_{1,proj} \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \times Homography \tag{5.10}$$

where $x_{1,proj}$ and $y_{1,proj}$ are the projected coordinates of a keypoint from the first image to the second image, x_1 and y_1 are the original coordinates of a keypoint in the first image, and $Homography$ is the 3×3 homography matrix that shows the transformation from the first image to the second image and is given in the dataset.

In the next step, the distances of the projected coordinates of the keypoints from the first image and the coordinates of the keypoints from the second image are calculated. If the Euclidean distance measurement of a projected keypoint and the keypoint from the second image is less than a threshold M the matching is considered to be correct. Otherwise, the matching is a false positive. We report our results with multiple values of the threshold M (1, 3, and 5 pixels) for evaluation of the proposed methods. A lower M represents a more strict evaluation metric. In addition to SIFT and BRISK descriptors, we provide the results of three early fusion techniques for comparison. The first early fusion technique is based on a raw fusion of BRISK and SIFT descriptor values. For evaluation of this technique we use Euclidean distance measurement. The second and third early fusion techniques are selecting the minimum distance and maximum distance between the Hamming distance measured for BRISK and the Euclidean distance measured for SIFT. Since Hamming distance and Euclidean distance measurements result in different ranges of values, they should be converted to the same range for minimum and maximum distance fusion. We compute the histogram of distances on the training folds for both Hamming and Euclidean distances and scale the Hamming distance measurement values to have the same mean with the Euclidean distance histogram. Our experiments show that scaling the Euclidean distance measurement values to Hamming distance range leads to similar results. The mAP results of our experiments for $M = 1$ in comparison with SIFT and BRISK descriptors are shown in Fig. 5.6 and Table 5.2. The values presented in Fig. 5.6 include the minimum, average, and maximum values of the test results on the 5-fold cross-validation. The descriptors are generated from the patches around the selected keypoints and the mAP is measured over the test sets. The highest average mAP is achieved by BRISK-SIFT which is 46.2%. This value shows a 14.6% improvement over the BRISK local descriptor algorithm (40.3%) and a 17.9% improvement over the SIFT local descriptor algorithm (39.2%). In terms of average mAP, BRISK-bSIFT and nbBRISK-SIFT have achieved 41.5% and 43.1%, respectively.

For $M = 3$ (Table 5.3), the highest average mAP is for BRISK-SIFT which is 71.0% and is 21.6% higher than the average mAP for SIFT (58.4%) and 15.8% higher than the average mAP for BRISK (61.3%). Our lowest average mAP among the three proposed methods is 63.8% (BRISK-bSIFT) which outperforms SIFT and BRISK by 9.2% and 4.1%, respectively.

The highest average mAP for $M = 5$ (Table 5.4) is measured by using BRISK-SIFT (75.5%) which is 21.8% higher than that of SIFT and 15.6% higher than the average mAP attained by BRISK.

Tables 5.2, 5.3, and 5.4 present the results for $M = 1, 3$, and 5, respectively. Figures 5.6, 5.7, and 5.8 illustrate the graphical representation of the values in Tables 5.2, 5.3, and 5.4. In all cases, our three proposed

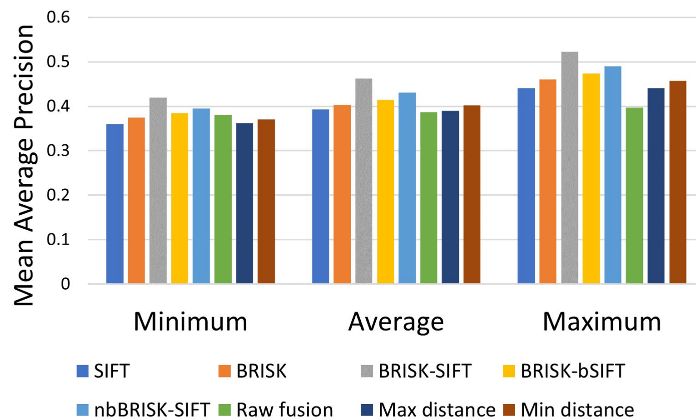

Figure 5.6: Comparison of mAP measurement results of our proposed methods with SIFT and BRISK descriptor algorithms. The threshold distance value for evaluation is $M = 1$.

Table 5.2: mAP measurement results of local descriptor algorithms with M=1

Descriptor	Minimum	Average	Maximum
SIFT	0.361	0.392	0.441
BRISK	0.374	0.403	0.460
Raw fusion	0.381	0.387	0.397
Min distance	0.371	0.402	0.457
Max distance	0.362	0.390	0.441
BRISK-SIFT	0.419	0.462	0.522
BRISK-bSIFT	0.385	0.415	0.473
nbBRISK-SIFT	0.395	0.431	0.489

Table 5.3: mAP measurement results of local descriptor algorithms with M=3

Descriptor	Minimum	Average	Maximum
SIFT	0.541	0.584	0.644
BRISK	0.599	0.613	0.629
Raw fusion	0.575	0.585	0.594
Min distance	0.589	0.602	0.617
Max distance	0.569	0.584	0.598
BRISK-SIFT	0.699	0.710	0.732
BRISK-bSIFT	0.622	0.638	0.651
nbBRISK-SIFT	0.648	0.659	0.678

methods outperform the early fusion methods and BRISK and SIFT algorithms operating independently.

Fig. 5.9 shows examples of matching pairs for several image sets from the HPatches dataset. From each image set, the matching algorithms are applied to the first image and the third image of the set. After

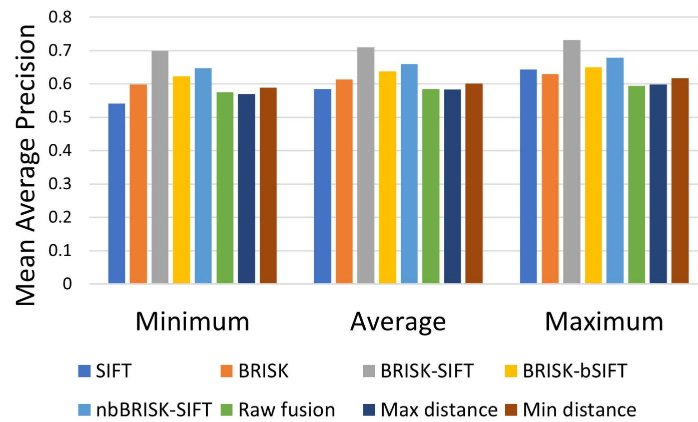

Figure 5.7: Comparison of mAP measurement results of our proposed methods with SIFT and BRISK local descriptor algorithms. The threshold distance value for evaluation is $M = 3$.

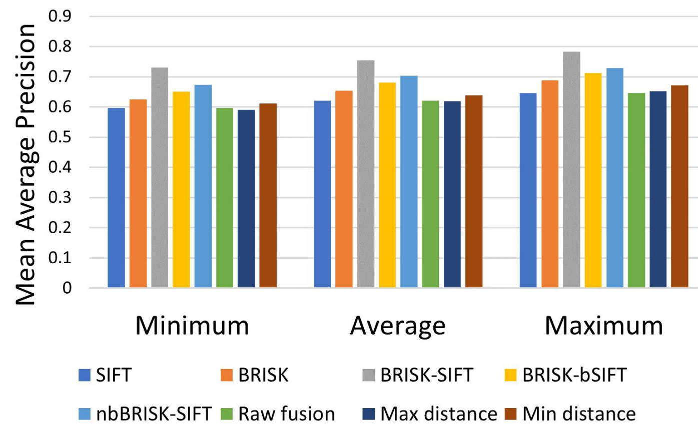

Figure 5.8: Comparison of mAP measurement results of our proposed methods with SIFT and BRISK local descriptor algorithms. The threshold distance value for evaluation is $M = 5$.

Table 5.4: mAP measurement results of local descriptor algorithms with M=5

Descriptor	**Minimum**	**Average**	**Maximum**
SIFT	0.597	0.620	0.646
BRISK	0.625	0.653	0.688
Raw fusion	0.597	0.620	0.646
Min distance	0.611	0.638	0.671
Max distance	0.591	0.619	0.652
BRISK-SIFT	0.730	0.755	0.782
BRISK-bSIFT	0.651	0.681	0.713
nbBRISK-SIFT	0.674	0.703	0.729

computing the matching distances using each descriptor algorithm, the matches corresponding to the 500 lowest computed distances of each method is shown in Fig. 5.9. The red lines represent incorrect matches and the green lines represent the correctly found matches. The example pairs from the Castle, Bark, Greenhouse, and Asronautis image sets are shown in Fig. 5.9 (a) and the example pairs from the Home, Artisans, Woman, and Calder image sets are shown in Fig. 5.9 (b). As an example, our three proposed methods attained more correct matches than BRISK in the Castle image set. On the other hand, the fusion results are more similar to BRISK and show a higher number of correct matches than SIFT in the Bark image set. The number of correct matches shown in Fig. 5.9 shows the positive effect of fusion with respect to individual descriptor algorithms.

Another commonly-used evaluation on the HPatches dataset is using the extracted patches provided by the HPatches benchmark to measure mAP. The patches are extracted from all images in the dataset. For this experiment, we learn the weights for each of the methods on the Brown dataset. We use the Liberty image set for learning the weights for our three methods for evaluation.

Fig. 5.10 shows the mAP measurement over all patches in the HPatches dataset. As shown in Fig. 5.10, BRISK-SIFT and nbBRISK-SIFT attain higher accuracy than SIFT and root SIFT [138], which is an improved version of the SIFT algorithm. In addition, BRISK-bSIFT shows a better result than BRISK which is a binary algorithm. We also report the result of TCDesc [31] which is a deep learning based method for patch description in this experiment. Deep learning methods such as TCDesc achieve higher mAP than the handcrafted methods, but they usually have a heavy requirement for storage and a GPU platform to be comparable with non-deep learning based methods in terms of execution time [25], [140]. As an example in [31] the average number of matched pairs per second is about 12.7 on a GTX 2080 Ti GPU for TCDesc, while the same metric for SIFT is 29.49 on an Intel Core i7-3770K CPU platform [25].

5.4.3 Experimental results and measurements on Brown dataset

The descriptor combination techniques in literature have reported their results using various datasets and in multiple applications, and in some cases the details of the implementations are not provided. As a result, comparing all the descriptor fusion methods in one benchmark is not feasible. For comparison, we choose LMBD [28] which has conceptual similarity with one of our proposed methods and has reported the results on commonly-known benchmarks. In this section, we present the experimental results on the Brown (Photo tourism) dataset [139]. The common evaluation metric for this dataset is false positive rate (FPR) at 95% of recall value. Similar to other work [11], [28], we use the Liberty image set for training and report the test results on the Yosemite and Notre Dame image sets in Table 5.5. In addition to absolute FPR values, we also report the improvement percentage with respect to the base algorithms for LMBD and our three methods.

As shown in Table 5.5, the absolute FPR value of LMBD is more than our three proposed methods. This is not surprising as the four base descriptors that were combined in [28] are learning-based descriptors and they all have less FPR than SIFT and BRISK. However, our proposed methods show more improvement with respect to the base descriptors due to the more complementary information extracted from the baseline descriptor algorithms. As an example, our proposed BRISK-SIFT shows 11.18 points (33.0%) improvement with respect to SIFT (which is the base descriptor with lower FPR) for the Yosemite image set while LMBD shows only 4.55 points (26.78%) improvement towards RFD_R which has the lowest FPR. If we compare the

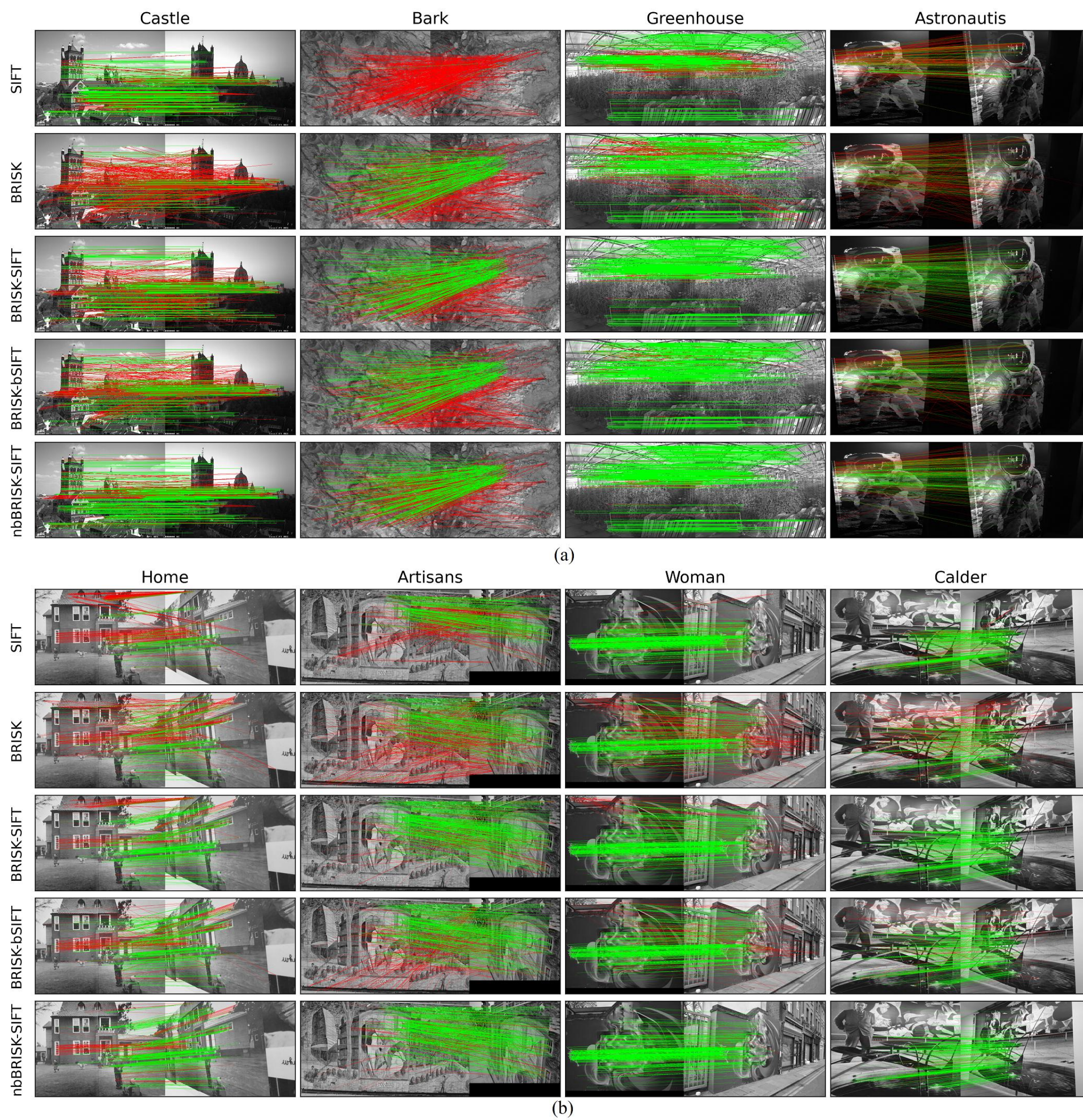

Figure 5.9: The visualization of 500 matches with the lowest computed distances in pairs of images from (a) Castle, Bark, Greenhouse, Astronautis and (b) Home, Artisans, Woman, and Calder image sets from the HPatches dataset [11]. The green lines demonstrate the correct matches and the red lines represent incorrect matches. The number of observable correct matches of our three proposed methods (BRISK-SIFT, BRISK-bSIFT, and nbBRISK-SIFT) is higher than the SIFT and BRISK algorithms in most cases. These images are best viewed in color and zoomed in.

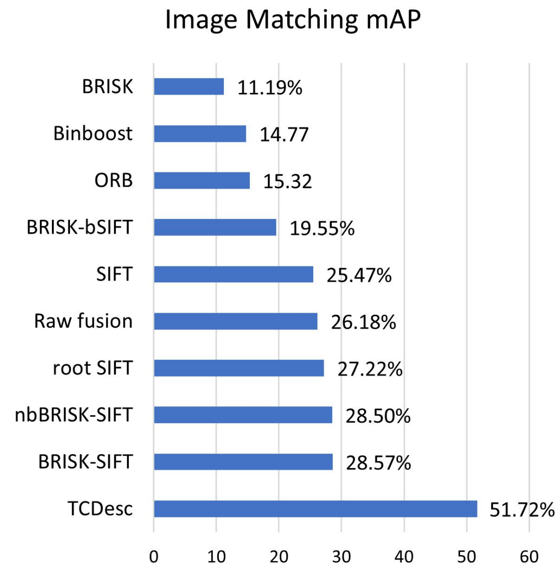

Figure 5.10: Experimental results on patch matching benchmark of the HPatches dataset.

improvement over the base descriptors with the highest FPR, our BRISK-SIFT method shows 50.5 (68.99%) improvement with respect to the BRISK descriptor while LMBD shows 10.44 points (45.63%) improvement with respect to BinBoost. The results in Table 5.5 show that the base algorithms that we have used in our proposed method have more complementary information in comparison with the base algorithms in [28]. The only method which does not show improvement with respect to the baseline algorithm is BRISK-bSIFT (with respect to SIFT) which is due to the loss of information when binarizing the SIFT descriptor.

5.4.4 Experimental results and measurements on Oxford dataset

In this section, we present our experimental results on the Oxford Affine Covariant Regions dataset [111]. Fig. 5.11 shows the comparison of our three proposed fusion methods with LMBD [28] on four image sets of the Oxford dataset. We use the same experimental setting as [28] for fair comparison. For each image set, the recognition rate (number of correct matches divided by number of selected matches) is computed. As shown in Fig. 5.11, our three proposed fusion methods attain higher recognition rate than LMBD in most cases in the presence of rotation and scale (Boat), blur (Trees), viewpoint (Wall) and JPEG compression (UBC) transformations. LMBD performs better on the last images of each set. However, the mean recognition rate of our methods outperforms LMBD in most cases. The mean of the recognition rate for LMBD is close to our methods for the UBC image set. This shows that the methods have similar performance for JPEG compression. However, our methods outperform LMBD on other test cases.

5.4.5 Discussion

Since each local descriptor algorithm can extract complementary information and features that may not have been extracted using the other algorithms, combining the descriptor vectors leads to higher accuracy. If the local descriptor algorithms extract similar information from the patch, the accuracy enhancement is

Table 5.5: False positive rate at 95% recall on Brown dataset

Absolute Results		
Descriptor	**Yosemite**	**Notre Dame**
SIFT	33.9	28.6
BRISK	73.2	74.9
Raw fusion	33.83	28.51
BRISK-SIFT	22.7	20.7
BRISK-bSIFT	55.0	56.2
nbBRISK-SIFT	26.9	23.3
RFD_G [132]	17.62	12.49
RFD_R [132]	16.99	13.23
BGM [133]	21.11	15.99
BinBoost [27]	22.88	16.90
LMBD [28]	12.44	9.52
TCDesc* [31]	1.28	0.33
Improvement		
Descriptor	**Yosemite**	**Notre Dame**
LMBD (w.r.t. RFD_G)	5.18 (29.40%)	2.97 (23.78%)
LMBD (w.r.t. RFD_R)	4.55 (26.78%)	3.71 (28.04%)
LMBD (w.r.t. BGM)	8.67 (41.07%)	6.47 (40.46%)
LMBD (w.r.t. BinBoost)	10.44 (45.63%)	7.38 (43.67%)
BRISK-SIFT (w.r.t. BRISK)	50.5 (68.99%)	54.2 (72.36%)
BRISK-SIFT (w.r.t. SIFT)	11.18 (33.0%)	7.89 (27.6%)
BRISK-bSIFT (w.r.t. BRISK)	18.2 (24.86%)	18.7 (24.97%)
BRISK-bSIFT (w.r.t. SIFT)	↓ 21.08 (38.37%)	↓ 27.64 (49.19%)
nbBRISK-SIFT (w.r.t. BRISK)	46.3 (63.25%)	56.6 (70.83%)
nbBRISK-SIFT (w.r.t. SIFT)	6.97 (20.59%)	5.25 (18.39%)

*TCDesc is a state-of-the-art deep learning method reported for comparison.

negligible.

By combining the descriptor vectors directly, the number of elements in the final combined descriptor vector increases and becomes more than the individual descriptor vectors. As a result, more elements are similar if the patches are from a correct match. Similarly, there will be more differing elements if the patches are less similar to each other. However, since the type of descriptor vector elements is different (binary or floating-point), without learning a scaling factor, the raw fusion of descriptors cannot always achieve better results than the descriptors operating independently. Minimum distance selection leads to missing the cases that one of the descriptors correctly determines that patches are not similar. On the other hand, maximum distance selection result in missing the cases that one of the descriptors correctly calculates a lower distance value for two similar patches. Therefore, these early fusion techniques do not always result in a higher mAP measurement.

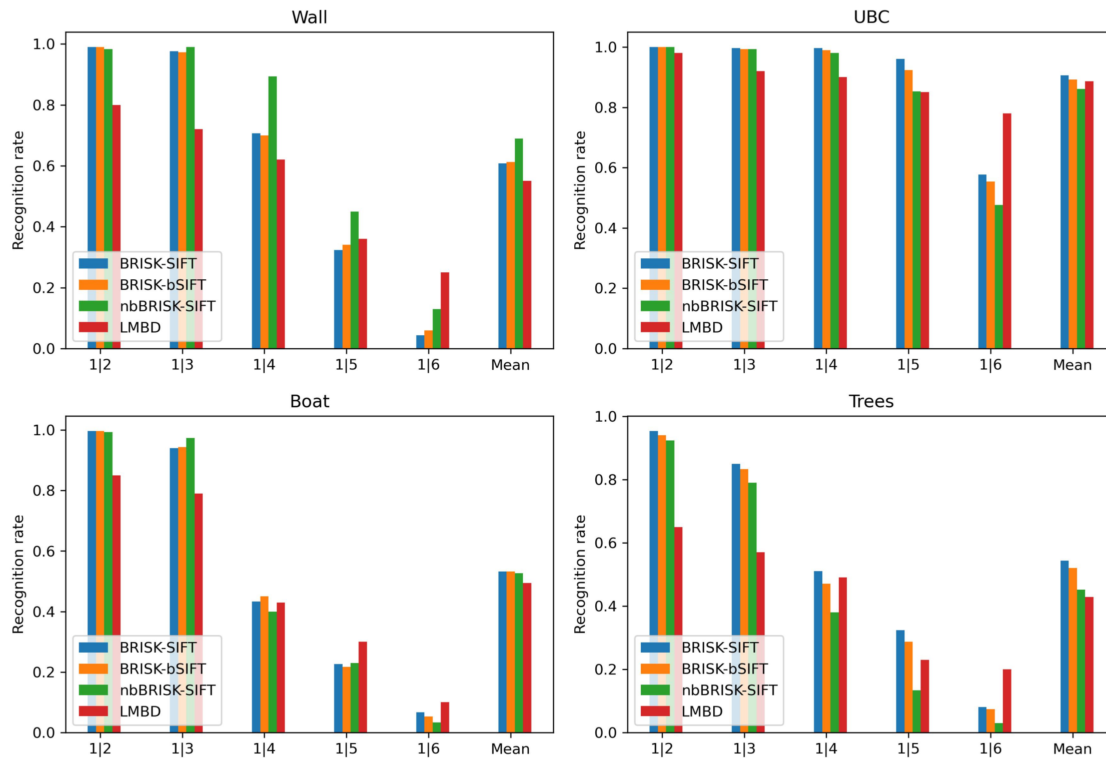

Figure 5.11: Recognition rate comparison of our proposed methods with LMBD on four image sets of the Oxford Affine Covariant Regions dataset [111].

In BRISK-bSIFT (an example of binary-fusion), a major part of the information extracted by SIFT is discarded as the values are quantized to 0 and 1. However, the threshold for binarizing the SIFT descriptor vector is chosen such that the overall accuracy measured on the training data is enhanced. Since the HPatches training set consists of a variety of scenes and models, the chosen threshold for quantization also works well for achieving a higher mAP measurement on the test set as shown in Tables 5.2 to 5.4.

When using the Euclidean distance measurement for descriptor vectors with a higher number of elements, and if the original binary vectors are similar to each other, the overall distance does not change substantially. However, for non-similar patches, adding the weighted binary descriptor as in nbBRISK-SIFT (an example of non-binary-fusion) improves the discrimination among the final descriptors and increases accuracy as shown in Tables 5.2 to 5.4.

When the distance metric is used for local descriptor algorithm combination, the results of each algorithm are directly used for the final fusion result as shown in (5.3). If both local descriptor algorithms determine that the patches are similar (distance values are lower), the combined distance metric decreases as well. On the other hand, if both local descriptor algorithms determine that the patches are not similar the combined distance value increases. If one algorithm shows less similarity measurement and the other one shows more similarity, the final result is determined from (5.3). The weights that are learned for BRISK-SIFT (an example of weighted-fusion) indicate the importance of each descriptor based on the experience learned from the training dataset.

Evaluation on multiple benchmarks reveal that our proposed methods achieve higher accuracy than the

baseline algorithms. Our weighted-fusion method learns one weight per descriptor algorithm plus a bias term while binary-fusion and non-binary-fusion learn only one parameter each. The fewer number of parameters result in less dataset dependency than other learning-based methods. In addition, the learning on the Brown dataset and the evaluation on the HPatches and Oxford dataset show that the learned parameters are general and can be used for other datasets. The Oxford benchmark evaluation shows the superiority of our method over LMBD in full-size images. The results on the Brown dataset compares the complementary information gain of SIFT and BRISK over the ones chosen by LMBD [28]. BRISK-bSIFT performs lower than the SIFT algorithm in HPatches patch matching and Brown benchmarks. However, the experiments on 5-fold cross validation on the HPatches dataset show that all three methods attain higher mAP measurement with respect to base algorithms in the presence of rotation and scaling, and with higher number of keypoints in the detection stage. Large numbers of keypoints is essential in applications such as aerial image matching [141].

In terms of computational complexity and required storage, our proposed methods require more memory and computations than the base algorithms. BRISK-SIFT computes two descriptors and two distances. BRISK-bSIFT results in a 640-bit binary descriptor vector (512 bits for BRISK and 128 bits from SIFT) and nbBRISK-SIFT generates 640 floating-point values. However, they are generally much less resource intensive compared to deep learning based methods. Parallel implementation of the descriptors on platforms such as FPGAs can compensate for the additional required time and computation.

5.5 Conclusion

In this work, we proposed three methods for fusion of two handcrafted descriptors. The proposed methods are weighted-fusion, binary-weighted-fusion, and non-binary-fusion which are defined in Table 5.1. As a case study, we performed experiments on combining the BRISK and SIFT local descriptor algorithms. Our results demonstrate that by fusing BRISK and SIFT local descriptor algorithms we can achieve a higher mAP on the test set than by using each of the algorithms individually.

In future work, we plan to extend our experiments using our three proposed methods to include other binary and non-binary algorithms and to combine handcrafted local descriptor algorithms with other machine learning models. We also plan to develop a hardware implementation of our fusion methods which can accelerate vision-based industrial instruments.

Chapter 6

Improving Handcrafted Image Matching Accuracy Using Learned Convolutional Filters and Processing Speed by Hardware Design

This paper presents a new filter design based on a shallow convolutional neural network for prefiltering images to improve image matching accuracy. Hill climbing, a commonly-used search optimization algorithm, is used to train a shallow and computationally efficient convolutional network to be deployed at the early stage of an image matching pipeline. The proposed technique for prefiltering the images using a shallow CNN is also applied to the fusion of two handcrafted descriptor algorithms (SIFT and BRISK) based on our previous work to further improve accuracy. In addition, we introduce an FPGA-based hardware design for the fusion of SIFT and BRISK descriptor algorithms to accelerate image matching application. The proposed SIFT description hardware is a combinational circuit implemented as a set of parallel modules to accelerate the overall description process. Our proposed hardware architecture for image matching acceleration comprises two layers of convolutional network, BRISK descriptor, SIFT descriptor and a module to combine the SIFT and BRISK descriptors. After demonstrating the performance of our proposed model on the HPatches dataset, we apply our image matching method to the challenging task of badger identification, which benefits from image matching due to the differences in the characteristics of individual badger faces. Our combination methods with convolutional prefiltering achieve a higher F-score than both the SIFT and BRISK baseline algorithms.

6.1 Introduction

Image matching is a process for finding the correspondences of the same physical points in two images taken from different angles or in different times. The pixels in an image that have characteristics which distinguish them from other pixels around them are known as keypoints. The purpose of image matching is to find

the matches of keypoints of one image (target image) to another image (reference image). Image matching results are a base for many practical applications such as structure from motion [113], 3D reconstruction [115], object recognition [116], and visual simultaneous localization and mapping (SLAM) [114].

Image matching algorithms can be categorized into two groups: learning-based and handcrafted [4]. The handcrafted algorithms use features such as intensity patterns of pixels in a local area of an image to differentiate between keypoints in the image. The extracted features are called image descriptors and are used to find similar keypoints between two images. On the other hand, the learning-based algorithms benefit from a training dataset to learn parameters for tuning descriptor models for a specific application and extracting descriptor vectors, which result in a higher image matching accuracy.

One of the obstacles of using image matching in practical applications is the high number of computations. Although some learning-based methods such as L2-Net [18], CNDesc [120], D2-Net [121], and Superpoint [122] achieve better accuracy by using deep neural networks, they require higher number of operations and therefore, demand more time to process which makes them unsuitable for real-time or low power applications. In addition, Efe et al. [124] and Jin et al. [125] show that the performance gap between handcrafted and deep learning-based methods is not significant after tuning specific parameters for evaluation.

A high number of operations generally results in slower computation in image matching algorithms. One solution to increase the speed of image matching is to implement the algorithm in parallel on platforms such as Field Programmable Gate Arrays (FPGAs). FPGAs provide low power and high speed design solutions which can be used efficiently in embedded systems.

In this work, we add a shallow learned convolutional filter to augment handcrafted algorithms and show that the mean Average Precision (mAP) of image matching improves. By using a convolutional network with shallow layers (instead of deep layers), we keep the number of computations lower than in a deep network while increasing the accuracy of a handcrafted descriptor algorithm. In addition, we propose an accelerated hardware design for implementing the proposed prefiltering method in combination with our previous work [41] on an FPGA platform.

Computer vision has environmental applications such as animal detection, identification, population control and wild life surveillance [142]–[144]. In recent years, many of the classical computer vision applications, such as facial recognition and individual re-identification has extended to animals as well. A few examples of this new focus can be seen in research work for re-identification of birds [145], seals [146], pumas [147], ursids [148], tigers [149], and chimpanzees [150]. In this work, as a case study of our proposed methods, we apply our work (shallow convolutional filter and handcrafted descriptor) on the badger identification problem to distinguish different badgers and identify them. The task of identification of badgers based on their facial markings is challenging due to the facial similarities of individual badgers. However, using image matching, we can use the correspondences of various characteristics of their faces to match the same badgers in multiple images.

6.2 Related work

6.2.1 Learning-based methods for image matching

There are many learning-based approaches proposed for image matching in the literature. We can categorize the learning-based algorithms to the ones that are based on deep neural networks and the ones that are not.

Tian et al. [18] and Wang et al. [120] propose models for the description stage of image matching based on deep neural networks. Dusmanu et al. [121] and DeTone et al. [122] propose an end-to-end convolutional network which includes both the detection and description stages of image matching. Although deep learning based approaches have achieved noticeable results on image matching datasets, they require a large number of computations which decreases the execution speed for practical applications with time constraints. Other learning-based approaches such as LMDB [28] and BEBLID [26] learn a smaller number of parameters for improving image matching accuracy.

We discuss the fusion of descriptor algorithms using learning parameters in our previous work [41]. In [41], we propose three methods for combining descriptors which embed complementary information for image matching. As an example, we use SIFT, an algorithm which generates non-binary descriptor vectors, and BRISK, an algorithm which generates binary descriptor vectors, as the base algorithms for combination. The three methods are called BRISK-SIFT (where the distance metric for each descriptor is used for combination), BRISK-bSIFT (in which SIFT is converted to a binary vector and is concatenated with BRISK), and nbBRISK-SIFT (in which BRISK is converted to a non-binary vector and is concatenated with SIFT). In [41], we show that the three fusion techniques lead to a higher mean average precision with respect to the original SIFT and BRISK algorithms, using cross-validation on the HPatches dataset. In this work, we propose convolutional filtering on images before detection and description, and we apply this method to our fusion techniques presented in [41].

6.2.2 Convolutional neural networks as learned image filters

Convolutional neural networks (CNN) have been an important part of the deep learning framework and are used in many applications of computer vision in recent years [151].

CNNs are usually comprised of several layers, and each layer has a number of sublayers. Each sublayer is defined by a kernel filter which has the same size for all sublayers in that layer. The size of this filter depends on the design of the CNN for a specific application. The input of each layer can be the input image (for the first layer) or the output of the previous layers. The output of each layer is called a feature map which is used as the input of the next layer. A CNN is defined by a set of parameters such as number of layers n_l, number of sublayers in each layer n_{sl}, input image size, and the kernel filter size of the sublayers in each layer.

For computation of the output of a layer, the kernel filters are convolved with the input feature maps of that layer. For convolution operation, the kernel filter is moved as a sliding window over the input image (or feature maps). At each position of the window, each weight of the kernel filter is multiplied with the pixel values of the corresponsing window on the input feauture map. Then, the summation of the multiplication values is computed. The summation result is the output of the sublayer and stored in the next feature map.

In a deep learning approach, CNNs include image filtering, feature extraction, and even decision making for classification task. In this work, we use a shallow CNN with a small number of layers and sublayers, only for filtering the image before keypoint detection. The main advantage of using a CNN in this work is that the weights of the filters are learned from the training data. By learning the weights using an optimization algorithm, we generate image filters that are tuned to increase the accuracy for a specific application such as image matching.

6.2.3 Hardware implementation of image matching algorithms

One of the important steps of feature-based image matching is the local description of image patches. Local descriptor algorithms extract features from a patch, which is a small window of pixels around a keypoint, and require a high number of repetitive computations on the pixels. As a result, implementing the descriptor algorithms based on parallel modules on platforms such as FPGAs has gained attention. There have been many recent works in the literature on implementing image matching steps on FPGAs. Sun et al. [33] implement the ORB algorithm [12], and Kreowsky et al. [152] and Li et al. [153] implement the SIFT algorithm on FPGA hardware. In our previous work, we implement the HOG [39] and BRISK [40] algorithms on hardware to increase the speed of these algorithms. Recent proposed hardware designs for local description, such as [33], [152], and [153], focus on speeding up the local description process of a single algorithm. Another level of parallelization is to implement and combine two or more algorithms in parallel in a way that increases the image matching accuracy.

In this work, we propose a hardware design which combines two local description algorithms. Our proposed hardware architecture includes learning-based convolutional filters to enhance the image before the detection stage, which improves the overall accuracy of image matching.

6.3 Learning-based convolutional filters

In this section, we introduce our method for improving the accuracy of image matching. There are many handcrafted filters introduced in computer vision literature. In some cases, increasing the contrast factor of an image may improve the matching results, while in other cases other filtering methods may have the same effect. In this work, we introduce a learning method to create filters to transform the input patch in a way that it is more suitable for image matching. We use convolutional neural networks (CNN) at the beginning of the image matching pipeline. We utilize a learning approach to compute the best CNN filters for image matching based on the training data. Figure 6.1 shows the block diagram of our proposed matching method. keypoint detection, patch description, and matching are common steps of any image matching algorithm. Our proposed CNN filter is be added before keypoint detection so that the filtered image is used for both detection and description.

In order to learn the weights of the filters, an optimization algorithm is used to modify the weights based on the matching output as shown in Fig. 6.1 (a). The optimization algorithm shown in Fig. 6.1 (a) is a part of the training procedure. After learning is completed, the optimization algorithm is not used in the deployment of the model for testing and evaluation (Fig. 6.1 (b)). In our proposed method, we cannot train the CNN using the back propagation algorithm which is commonly-used for training CNNs. The reason is that there is no differentiable path between the matching output to the filter stage in the matching pipeline. Chalup et al. [154] study using a hill climbing algorithm for training the weights of a small feed forward neural network with only two hidden neurons. In this work, we use a hill climbing algorithm [155] as a search optimization algorithm for tuning the weights of the CNN in Fig. 6.1.

In this work, various architectures of shallow convolutional neural networks are explored to achieve higher matching accuracy. We prefer simple models with a few number of layers for CNNs to limit the computational cost of the model. The best results from our experiments are attained using an architecture of two convolutional layers each having one sub-layer.

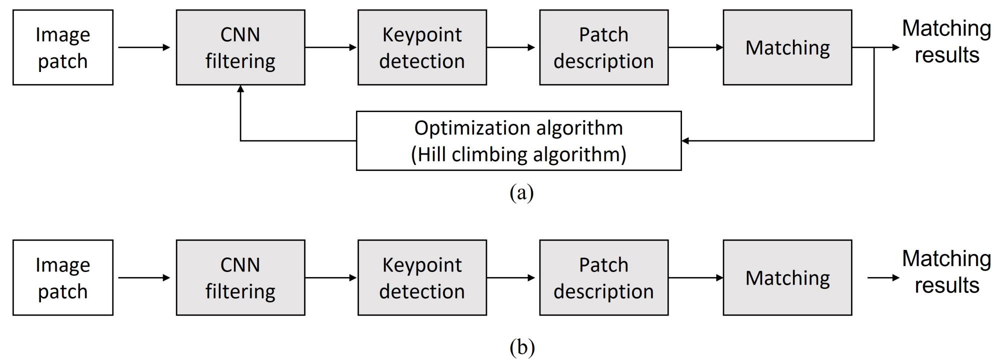

Figure 6.1: The block diagram of our proposed method. The training phase includes optimization using a hill climbing algorithm (a). The deployment phase uses the learned CNN weights for prefiltering the image patch before other image matching steps (b).

6.3.1 Training using a hill climbing algorithm

In this section, we review the training procedure of the hill climbing algorithm. A hill climbing algorithm is a greedy search optimization algorithm that proposes solutions in the neighborhood of the latest accepted solution at each iteration. The solutions in the same neighborhood are the ones that are close to each other in the search space. The new solution is applied to a loss function and a loss value is calculated based on that solution. If the loss value of the current iteration is less than the loss value of the previous iteration, the new solution is accepted and used to generate the neighborhood solutions of the next iterations. Otherwise, another solution from the neighborhood of the latest accepted solution is selected and evaluated in the next iteration.

In this work, we map the problem of training the CNN filter weights to the hill climbing algorithm. We use 5×5 kernels for each of the two layers of the CNN and the set of weights are the solutions generated from the hill climbing algorithm. The initial solution is generated randomly.

The steps of generating the filter solution are demonstrated for an example in Fig. 6.2. To create a neighborhood solution, a random 5×5 mask is generated where the value of each element is in the range of 0 to 1. Then, the mask elements are converted to -1, 0, and 1 by comparison with the two threshold values of 0.333 and 0.667. We assign the mask elements that are less than 0.333 to -1, between 0.333 and 0.667 to 0, and greater than 0.667 to 1 in a thresholded mask. After that, we create a learning mask based on the learning rate α and the thresholded mask as shown in Fig. 6.2. Based on the corresponding mask element, if the mask element is 0, we keep the corresponding element of the filter unchanged. If the mask element is 1, a constant value α is added to corresponding element in the filter. If the mask element is -1, the same constant α is subtracted from the corresponding element. According to this definition, each 5x5 filter solution has $3^{25} - 1$ neighborhood solutions (where -1 stands for the same solution in the case that no element changes). We stochastically evaluate randomly created solutions from each neighborhood as it is not practical to evaluate all possible solutions in a neighborhood.

Since hill climbing is a greedy algorithm it can become stuck in local minimum locations in the search space. To alleviate this limitation, if there is no change in the loss value in the last T iterations (T is chosen empirically and is set as 20 in our experiments), a completely new random filter is generated and the loss function is evaluated based on the new solution. In this way, if the new random filter is better than the

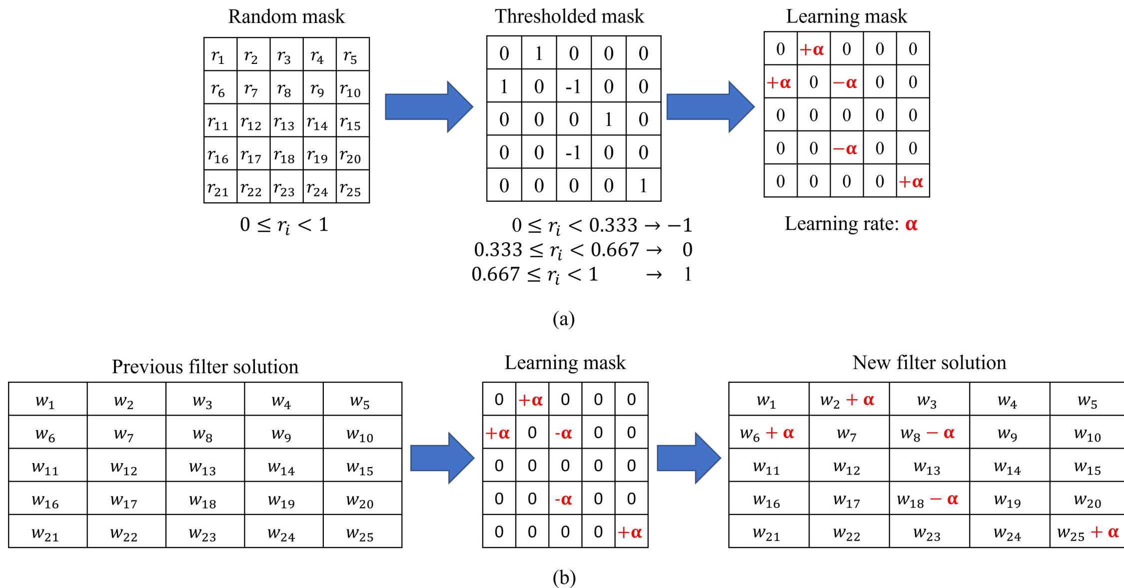

Figure 6.2: An example of the generation of the learning mask (a) and an example of the generation of the new filter solution based on the previous filter and the learning mask (b). r_i is a random value between 0 and 1, and α is the learning-rate.

current solution, the algorithm selects new solutions from the new neighborhood afterwards and escapes the local minima. If after 100 iterations there is no change in the loss value, the latest accepted solution is selected as the best solution and the output of the algorithm.

For evaluation of the loss function, first, the convolutional filter with the learned weights is applied to the training images. Then, the keypoints are detected and descriptor vectors are computed. Finally, we compute the mean Average Precision (mAP) metric on the training set. The loss value is defined as shown in (6.1).

$$Loss = \frac{1}{\text{mean Average Precision of a training batch}} \tag{6.1}$$

We use the HPatches dataset [11] for the training and evaluation using various CNNs to preprocess the images. The HPatches is a commonly used dataset containing 116 sets each having 6 images, which include image transformations such as illumination, rotation, scale and projection. We present the selection process for the parameters of our CNN model in section 6.3.2.

6.3.2 Parameter selection

The experimental results of our method for parameter selection is provided in this section. Figure 6.4 shows the effect of selecting different learning rate α on the learning process. In our training approach, the learning rate does not change in the training phase and is a constant after its value is selected in the initial experiments. Intuitively, a higher learning rate results in exploring other neighborhoods in consecutive iterations while a lower learning rate leads to searching in more local spaces within the current neighbourhood and towards the local minimum in that search space. As shown in Fig. 6.4, the loss value does not have a noticeable change

for $\alpha > 0.2$. Therefore, we set $\alpha = 0.2$ for our further experiments. Figure 6.3 shows our CNN architecture, and an example of an input image and the filtered image as a result of convolution by the learned CNN.

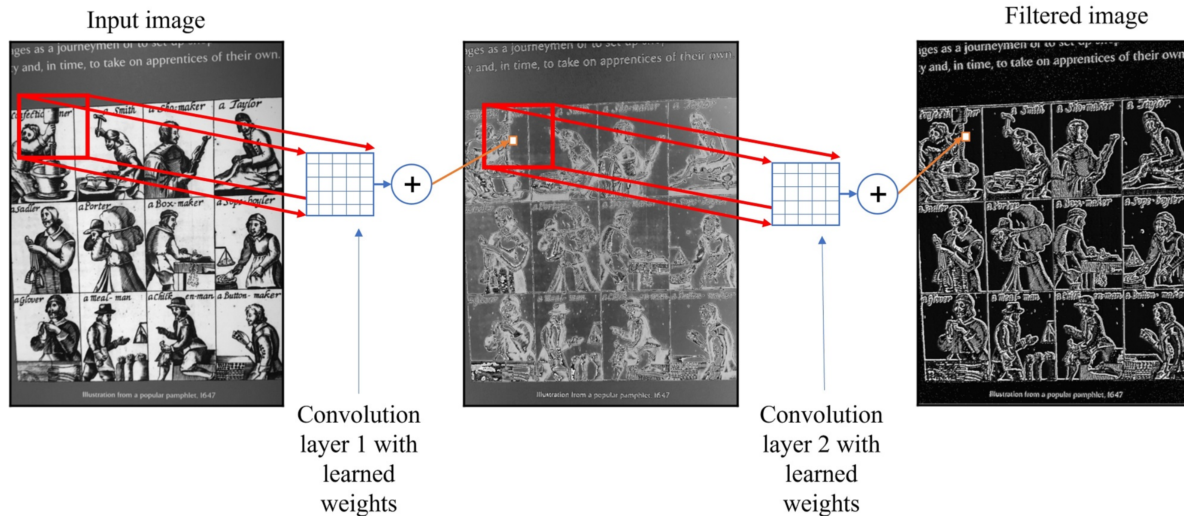

Figure 6.3: Architecture of our proposed shallow CNN and an example of the input image and filtered image. The example image is selected from the Apprentices image set of the HPatches dataset [11].

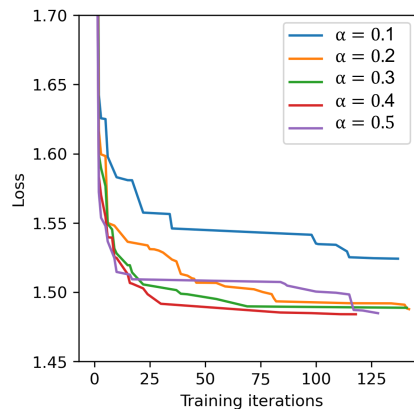

Figure 6.4: Loss curve over training iterations for various learning-rate (α) values. We choose $\alpha = 0.2$ for other experiments since the loss value does not have a noticeable change with $\alpha > 0.2$.

Figure 6.5 (a) shows the mAP curve on training iterations for various number of layers n_l (with the number of sublayers set as one) and Fig. 6.5 (b) shows the mAP curve on training iterations for various numbers of sublayers n_{sl} (with the number of layers fixed as 2). As shown in Fig. 6.5, after a number of iterations, the mAP curve saturates and it is more difficult for the algorithm to find a better solution. Based on the data presented in Fig. 6.5, we choose 2 layers, each having a sublayer, for our further experiments.

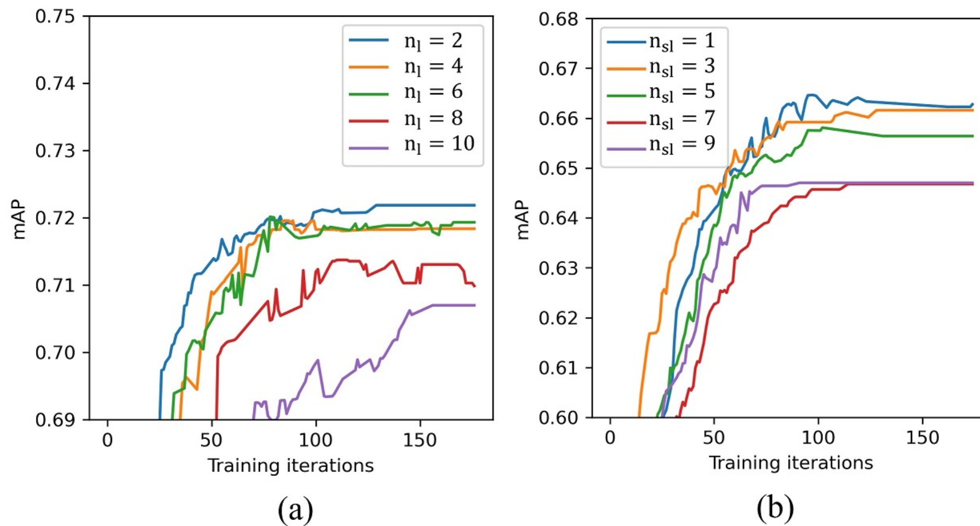

Figure 6.5: The mAP curves by changing the number of (a) layers n_l, and (b) sublayers n_{sl} of the CNN model. We choose two layers (n_l=2) each having one sublayer (n_{sl}=1) for the CNN filter.

6.3.3 Baseline comparison

We use 5-fold cross validation on the HPatches dataset [11] to compare the accuracy of our method with the baseline algorithms. In each image set, the first image is the reference image, and the homography matrix for transformation to other images is provided. For the 5-fold cross validation on the HPatches image sets, 4 folds are used for training and 1 fold is used for evaluation.

Figure 6.6 demonstrates the mAP value on the average of 5-fold cross validation on the HPatches dataset. The acceptance error margin is the minimum accepted Euclidean distance of the projected coordinates of a keypoint from the first image (reference) and the coordinates of the corresponding keypoint in other (target) images. We compute the mAP with the acceptance error margin of 1, 3, and 5 pixels, and compare the SIFT algorithm with conv-SIFT (convolutionally filtered SIFT) as shown in Fig. 6.6. In all cases, the result of the 5-fold cross validation shows that conv-SIFT attains higher mAP in comparison with the SIFT algorithm.

In addition to comparison with the SIFT algorithm, we apply our prefiltering CNN approach to the fusion methods for BRISK and SIFT algorithms as shown in our previous work [41]. Figure 6.7 compares the mAP of the algorithms with the same acceptance error margin of 3. We also compute mAP using HardNet [24], which is one of the commonly-used deep-learning based algorithms, in the 5-fold cross validation. As shown in Fig. 6.7, the combination of BRISK and SIFT using weighted distance when applied with convolutional prefiltering attains the highest mAP. We use a pretrained model of the HardNet which is not fine-tuned on the HPatches dataset. However, the results in Fig. 6.7 shows that by learning the weights for a shallow CNN, we can achieve a mAP comparable to HardNet, which typically generates good results.

Figure 6.8 shows five examples of detected matches using the SIFT algorithm with and without our proposed CNN filter method (conv-SIFT). Our convolutional prefiltering leads to the detection of fewer keypoints (by eliminating noisy and uncertain keypoints) such that the overall matching accuracy improves.

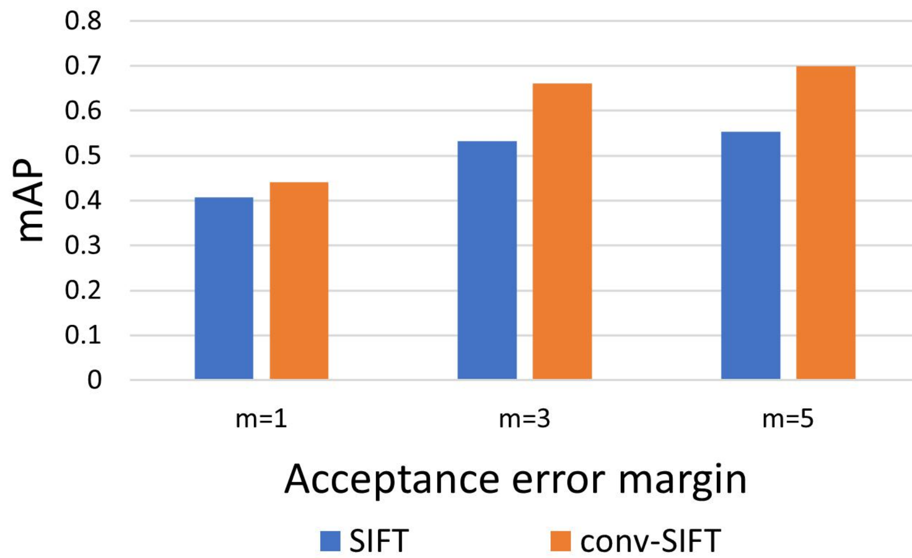

Figure 6.6: The comparison of SIFT and conv-SIFT algorithms using mAP on 5-fold cross validation on the HPatches dataset [11].

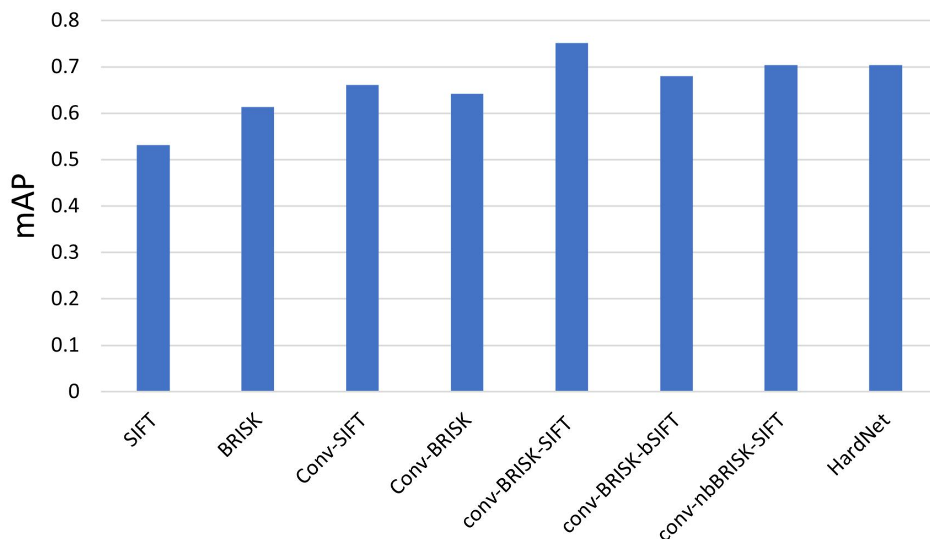

Figure 6.7: The mean Average Precision comparison on 5-fold cross validation on the HPatches dataset [11] for an acceptance error margin of 3.

6.4 Hardware implementation of prefiltered fusion algorithms

In this section, we introduce a hardware architecture that combines the CNN prefiltering described in section 6.3 and the fusion of BRISK and SIFT algorithms as described in [41] to accelerate the proposed image matching process. Based on the experiments in section 6.3, a shallow CNN with two convolutional layers

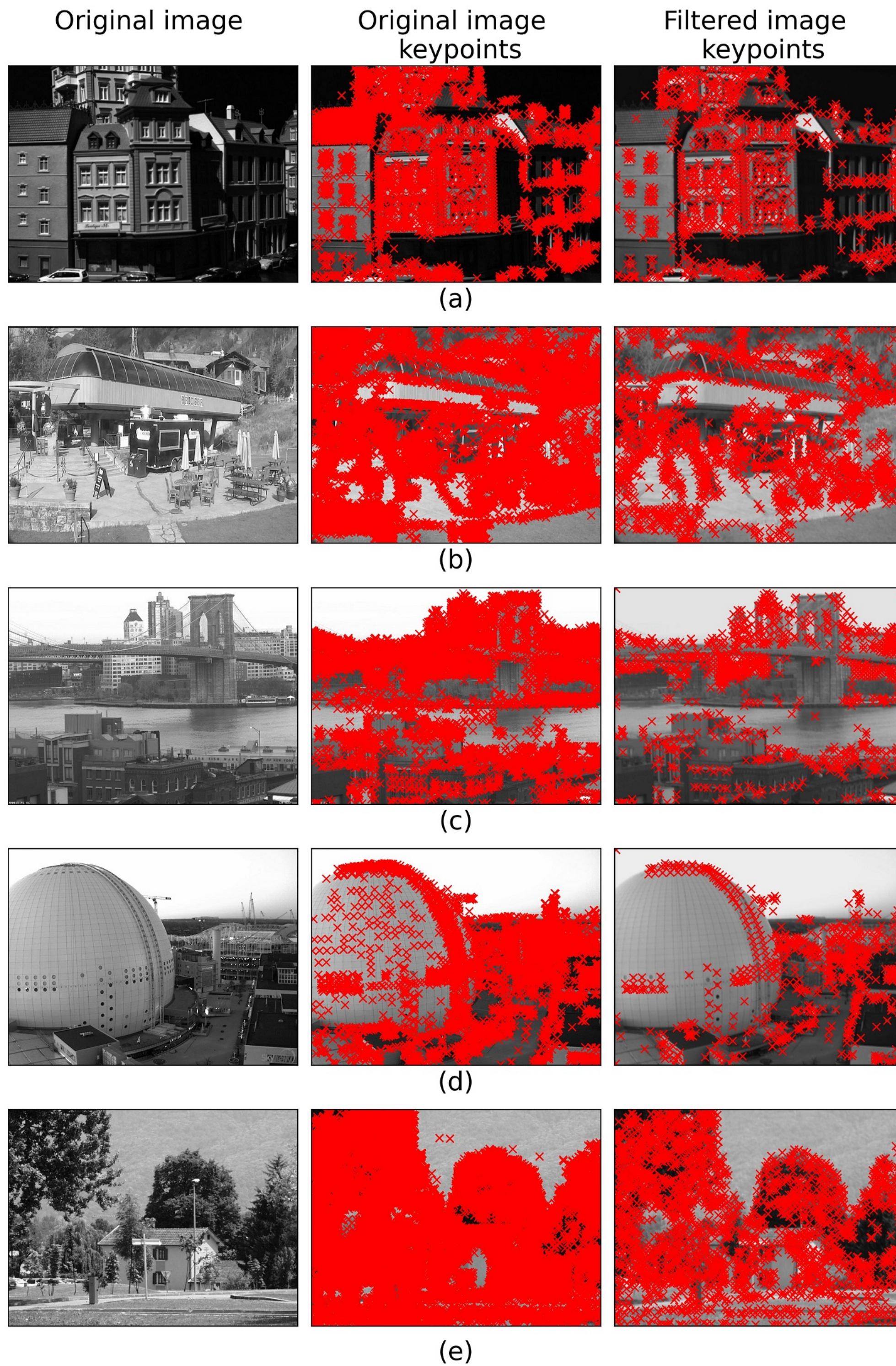

Figure 6.8: Example of matches detected in HPatches images using SIFT and conv-SIFT. The images are selected from the sets (a) Boutique, (b) Bridger, (c) Brooklyn, (d) Dome, and (e) Eastsouth from the HPatches dataset. The keypoints which result in higher matching accuracy are preserved in the filtered images.

each having one sublayer is selected as the preprocessing filter.

Figure 6.9 shows the functional block diagram of the various modules for computing the combined descriptors using our proposed methods. In this work, we use the FAST detector [10] which is a commonly-used keypoint detection algorithm to indicate if the input image patch contains a keypoint or not. If a keypoint is detected, the coordinate of the keypoint and the corresponding computed descriptor are the output and can be stored in memory for the next steps.

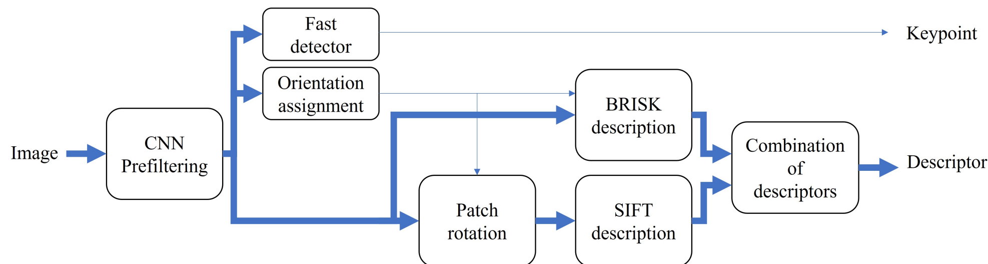

Figure 6.9: Functional block diagram of the hardware architecture

Figure 6.10 shows the overall architecture of our proposed design. There are three line buffers in this design which enable the computational modules to access each 5×5 pixel values in parallel. The first two line buffers each have $5 \times W$ registers where W is the width of the input image. The output of the first two line buffers are 5×5 windows of pixels which are fed into the convolutional filters. The weights of the filters are learned based on the experiments described in section 6.3. For computing the filtered image, we use shift and add operations instead of multiplication required for CNN convolution, as the filter weight values are precomputed and fixed from the training stage.

The third line buffer has $33 \times W$ registers. The output of this line buffer is a 33×33 window of pixels. From these the central 9×9 pixels are fed to the FAST detector module. The FAST detector module, BRISK description, and the orientation assignment module are implemented as explained in our previous work [40]. The orientation assignment module assigns a value between 0 to 35 to each patch, which indicates the main orientation of the input patch between 0 to 350 degrees in steps of 10 degrees.

After computing the main orientation of each patch, the central 16×16 pixels of the 33×33 window is rotated to compensate for rotation variations of the image patches. We use backward rotation coordinates to compute the orientation vector from the rotated window to the original window as proposed by Kreowsky et al. [152]. The rotation vectors are precomputed for each of the 36 orientations based on the output pixels rather than the input pixels. In this way, all rotated pixels have values from the input patch and there is no need for interpolation.

For SIFT description, the central 16×16 pixels are divided into sixteen (16) 4×4 cells in a 4×4 grid. The 4×4 cells are processed in parallel. First, the horizontal and vertical gradients are computed in parallel. Then, using the computed gradients in horizontal and vertical directions, the orientation and magnitude of each pixel is computed based on their gradients. Computing the gradients, orientation, and magnitude are the same steps as in the computations of the HOG algorithm and are discussed in detail in our previous work [39]. For the next step which is specific to SIFT, we flatten the orientations θ and magnitudes M of each cell into vectors of 16×1 as shown in Fig. 6.11 (a). Then, we generate an 8×16 binary matrix as shown in Fig. 6.11 (b) where each row corresponds to one of the 8 bins of histograms for that cell and each

column corresponds to the orientation θ of one pixel of the cell. Each row (bin) corresponds to a range of 22.5 degrees (overall 180 degrees) and if the orientation of the pixel is within a specific range, the value of that element in the matrix is set to 1. Otherwise, the value is set to 0.

In the next step, we compute the dot product of each row of the matrix in Fig. 6.11 (b) with the magnitude vector shown in Fig. 6.11 (a), as shown in Fig. 6.11 (c). The result of the dot product is the final value of the corresponding bin and the final histogram of a cell is obtained by concatenating the 8 bin values as shown in Fig. 6.11 (d). Our proposed design for SIFT description is a parallel combinational circuit and does not require a clock signal.

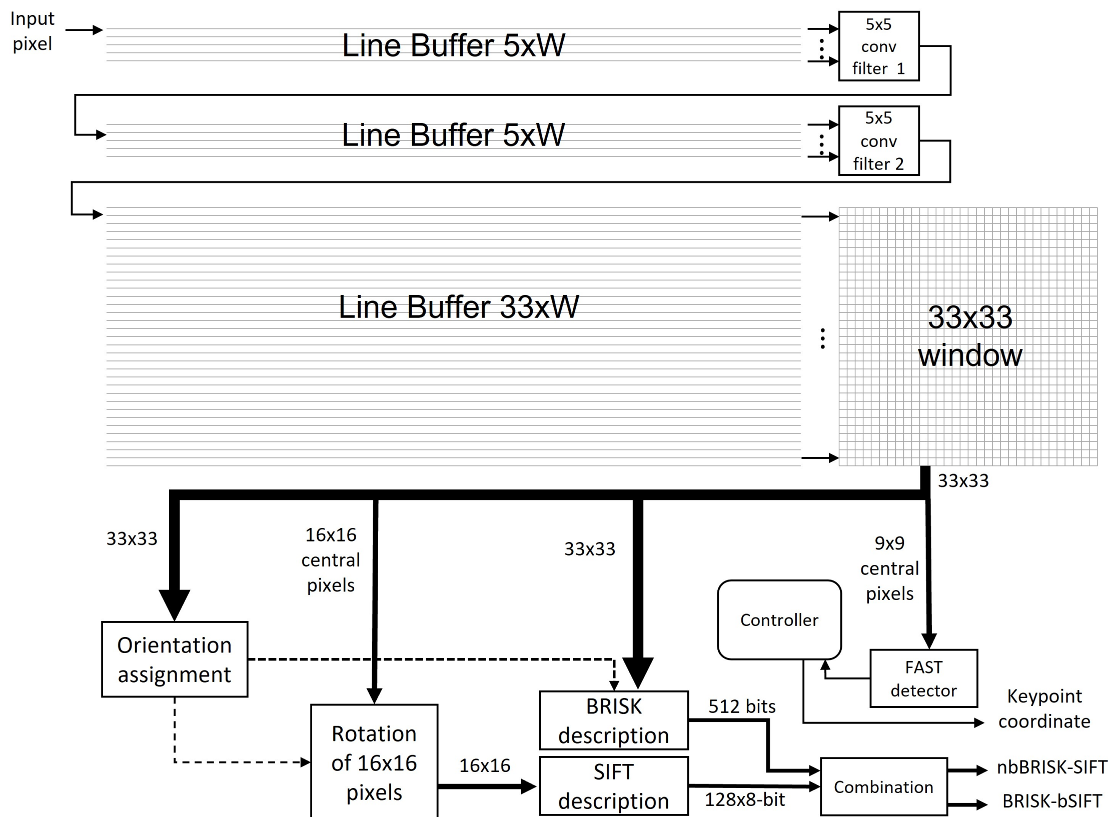

Figure 6.10: The overall architecture of our proposed hardware accelerator.

The next step of the SIFT description is normalization. For the normalization step, we use the procedure discussed in [39]. In this procedure, each bin value is divided by a constant so that the overall summation of the descriptor vector elements is bounded. First, the sum of the 128 SIFT elements is computed using a 7-level adder tree. Then, the sum value is divided by 128, by shifting the binary number to the right seven times. After that, the value is compared with empirically chosen values of 10, 20, 30, and 40, and based on the range of the value, an encoder generates three signals as MUX select signals. The three signals control all the 128 multiplexers as shown in Fig. 6.12. For each SIFT element, depending on the MUX selector, one of the multiplication results is selected as the output.

The multiplications shown in Fig. 6.12 are implemented using shift and add operations as the multiplicand values are constants. By further dividing the range for normalization, the resolution of normalization

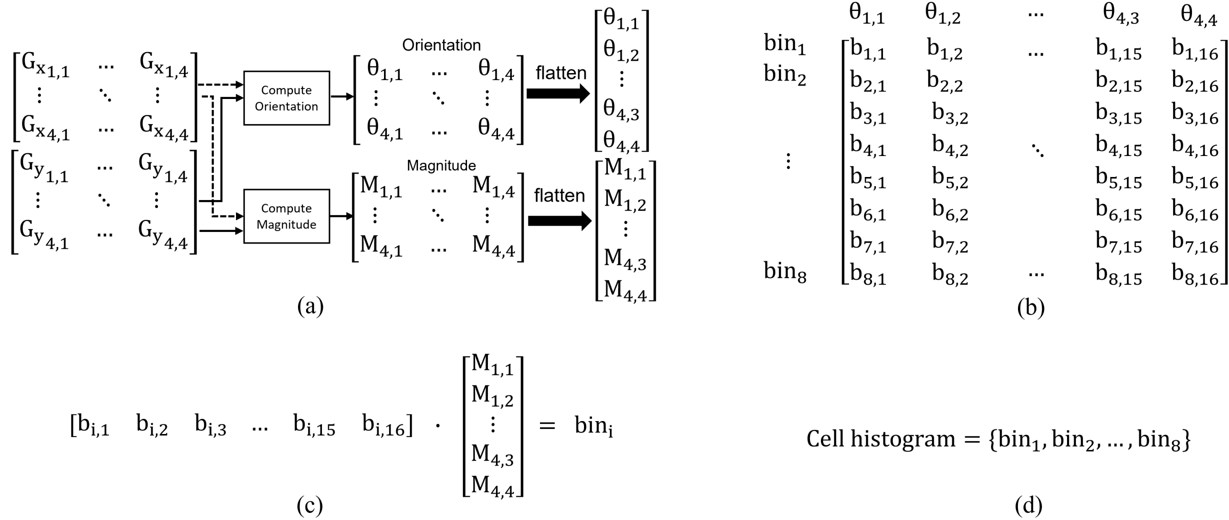

Figure 6.11: Steps of the SIFT descriptor generation. (a) Computations of the orientation and magnitude arrays for each 4×4 group of pixels. (b) The bin-angle matrix which facilitates combinational computation of the histogram. (c) The dot product for each row of the bin-angle matrix with magnitude vector. (d) The overall histogram of a 4×4 cell is the concatenation of 8 bin_i values.

increases which leads to a more accurate result. However, the design would require more hardware resources to implement the more accurate normalization. Therefore, we choose 5 ranges (less than 10, more than 40, and three ranges in between with steps of 10) as a trade-off between hardware resource usage and accurate normalization. The reason for having multiplication instead of division for normalization is that at the end, we multiply the normalized SIFT elements so that the values are in the range of 0 to 255. Therefore, by moving the multiplication operations to an earlier stage, we can multiply the values by $255/10 \approx 25$, $255/20 \approx 13$, $255/30 \approx 8$ and $255/40 \approx 6$ instead of having a division followed by a multiplication.

In the next step, the combination module concatenates the descriptors based on the BRISK-bSIFT or nbBRISK-SIFT methods depending on an input control signal. For BRISK-bSIFT, the SIFT descriptors are compared with a threshold and the result is concatenated with the BRISK descriptor. For nbBRISK-SIFT, the BRISK descriptor bits are used as the selectors of 512 2-input multiplexers. Each multiplexer selects from an 8-bit precomputed scaling value or 8-bit zero to output a scaled BRISK descriptor. The final non-binary BRISK vector is concatenated with the SIFT descriptor. The BRISK-SIFT combination is related to the matching stage and therefore, is not provided in this architecture. However, the combination of BRISK-SIFT can be implemented as a subsequent stage of the proposed architecture in Fig. 6.10.

6.4.1 Hardware evaluation

The FPGA resource utilization of our proposed method for a 1920×1080 image on a Kintex® Ultrascale™ (XCKU040) FPGA [109] is shown in Table 6.1. The hardware resources used for keypoint detection and description architecture of our design are independent of the image size. If the image size is scaled, only the number of line buffer registers increases. The main utilization of hardware resources is related to the description stage. The correctness of the design outputs was verified by comparison with the results of a

Table 6.1: Overall resource utilization of the proposed architecture on Kintex® Ultrascale™ FPGA (XCKU040) [109]

Resource	Utilization	Chip Utilization %
LUT	178863	26.96
LUTRAM	13512	4.6
BRAM	16	0.74

Table 6.2: Resource utilization of the modules of the proposed architecture

Resource	LUT	LUTRAM	BRAM	DSP
Line buffer 1	233	2400	0	0
Line buffer 2	233	2400	0	0
Line buffer 3	843	0	16	0
Convolutional layer 1	536	0	0	0
Convolutional layer 2	582	0	0	0
Detector	212	0	0	0
Orientation assignment	69116	0	0	0
Patch rotation	23188	0	0	0
SIFT descriptor	58383	0	0	0
SIFT Normalization	5064	0	0	0
BRISK descriptor	6589	0	0	0
Combination	256	0	0	0
Control unit	118	0	0	0
Synchronization registers	0	8712	0	0

Python-based software model.

Table 6.2 demonstrates the resource utilization of each module in our proposed design. We implement Line buffer 3 using BRAMs, and Line buffers 1 and 2 using LUTs to save LUT resources for other possible applications. The orientation assignment, which is implemented in parallel for achieving higher speed and lower latency, has the most resource utilization.

Our proposed architecture works with a maximum frequency of 105 MHz and computes each descriptor in 6 clock cycles after a keypoint is detected, which is indicative of the high speed of our design. The 6 clock cycles include 5 clock cycles for orientation estimation and one clock cycle for description.

Since other work have only focused on the hardware implementation of a single descriptor algorithm, we provide a comparison of our proposed architecture for individual algorithms as well. We have compared our BRISK descriptor hardware with other work in our previous work [40]. Table 6.3 demonstrates the comparison of hardware implementation metrics of our proposed method and other recent work for SIFT implementation. For patch rotation, we show the resource utilization comparing with the reported value by Kreowsky et al. [152]. Our work only requires 3 clock cycles to compute the SIFT description after the orientation assignment, and computes the SIFT descriptor with higher level of parallelism than other recent work such as [156], [157], [158] and [152]. In addition, the SIFT descriptor module in our work does

not require any DSP resources of the FPGA as all required multiplications are implemented using shift and add operations. The LUT resource utilization of our SIFT descriptor is higher than previous work (except Kreowsky et al. [152]) but requires no memory and DSP resources. The work by Kreowsky et al. [152] which operates at a maximum frequency of 175 MHz, requires more LUT resources in addition to DSP and memory bits than our proposed design. The Arria® 10 GX 1150 FPGA used in [152] has about three times more available on-chip memory and about two times more logic resources than the Kintex® Ultrascale™ platform, while other FPGAs shown in Table 6.3 have less available memory and LUT resources. However, the reported values in Table 6.3 are more related to the architecture designs than the FPGA platforms' capability.

Table 6.3: Comparison of hardware implementation metrics for the SIFT descriptor with other work

Work	**Year**	**FPGA**	**LUT**	**Memory**	**DSP/Multiplier****	**Frequency**	**Clk cycles**	**Processing time**
Vourvoulakis et al. [156]	2016	Cyclone® IV	46,854	897 kbits	47,524 M	25 MHz	102 *	4.08 us
Li et al. [153]	2018	Cyclone® IV GX	58,475	255 kbits	512 M	50 MHz	>14 *	0.28 us
Domenech-Asensi et al. [157]	2020	Virtex® 5	14,335	67,175 kbits	150 D	100 MHz	796	7.96 us
Chien-Hung Kuo et al. [158]	2021	Cyclone® IV GX	54,911	268 kbits	297 D	50 MHz	NA	NA
Kreowsky et al. [152]	2021	Arria® 10 GX 1150	117,066	14,589 kbits	138 D	175 MHz	28	0.16 us
Patch rotation			43,971	0	0			
Ours	2022	Kintex® Ultrascale™	86,640	0	0	105 MHz	1	9.5 ns
Descriptor			58,388	0	0		1	
Normalization			5,064	0	0		<1	
Patch rotation			23,188	0	0		<1	

* Estimated based on delays reported in architecture.

** M stands for multipliers and D stands for DSPs in this column.

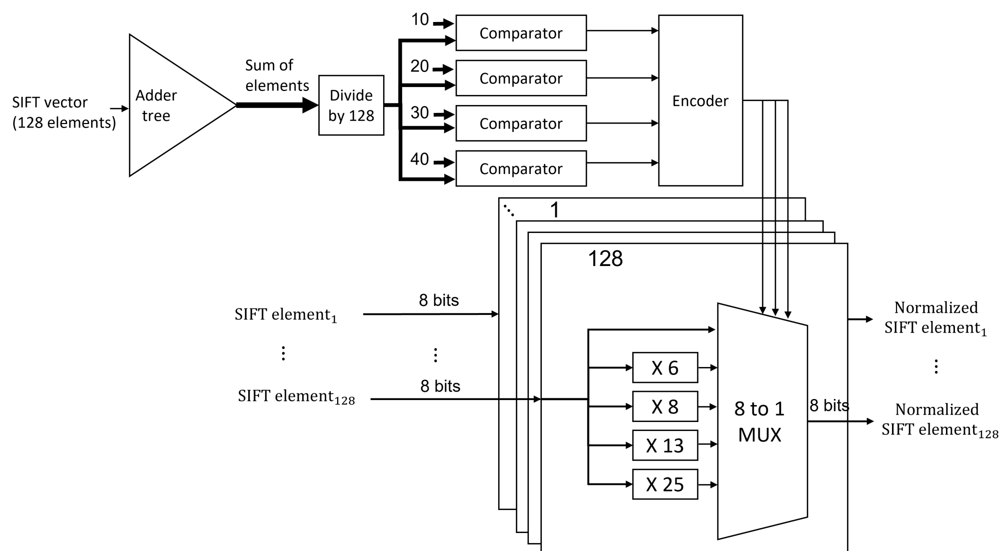

Figure 6.12: The block diagram of the normalization step for the SIFT descriptor vector.

6.5 Application to badger identification

Animal identification has attracted interest in recent years for applications such as the study of ecosystem, population dynamics, and behavioural ecology [142]. Although there is some work pertaining to specific animals such as bears [159], ursids [148], and mustelids [160], there has been less work focusing on identifying badgers. Villafañe-Trujillo et al. [160] examine 275 specimens of mustelids, a family of carnivorous mammals that include badgers, for individual identification based on the shape and color of their throat, and discuss that the differences are significant enough to allow discrimination of individual mustelids. Harrison [161] reports that individual American badgers (Taxidea taxus) may be identified by natural features of their dorsal head stripes. In [161], identification of the individual badgers is done by human observation. The author of [161] discusses that due to the small discriminatory features of patterns in head stripes, to identify a badger with this visual method, a close-up photograph or other close observation is necessary.

American badgers (subspecies jeffersonii) have been classified as endangered species in Canada. Many badgers die each year trying to cross busy highways, roads, and railway lines [162]. Klafki [163] presents a discussion on how an understanding of the ecology of badgers is integrated into habitat protection and restoration plans, including mitigation for current and future road infrastructure projects, thereby reducing impacts on this species. In order to build a safer environment for badgers, it is important to have a better understanding of their habitat, movements and population. Identifying the individual badgers helps the environmentalists to have a better sense of their location and normal daily activities. It is also important to identify them to facilitate their population management.

Gould et al. [164] use images taken by camera to estimate badger populations by detecting individual badgers based on the difference of their dorsal head stripes. Although they set motion triggered cameras in nature to detect the badgers, they identify the badgers manually and do not use any automated computer vision algorithms for the task. In this section, we apply our method to a set of images taken from badgers in the wild to evaluate our method for badger identification.

The image dataset for badger identification was acquired from the British Columbia Ministry of Transportation and Infrastructure. The images were taken using several cameras positioned in wildlife locations, between 1 May and 15 October 2019 at 14 culverts monitored in the trial and control areas of Highway 97 between 146 Mile and 150 Mile House in British Columbia, Canada. For the purpose of this work, we use the images which contain badgers only. In the next step, we extract the images in which the faces of the badgers are visible. Some examples of badger faces are shown in Fig. 6.14.

To prepare the dataset for evaluation, we categorize the badger images based on the location, date, and time that the images were taken. This procedure results in 38 sets of images, where each set is the images of the same badger. Some sets may only have one face image while others have more than one facial image. In order to evaluate our method, we compute the number of matched keypoints found between each of the two images in our prepared dataset. The goal is to increase the number of matches for images from the same set and decrease the number of matches for images that are not in the same set (and do not contain the same badgers). If the number of matches from images of the same set is higher than a threshold, the matching of the pair of images is a true positive match. The threshold can be set based on the requirements of the applications. Higher threshold results in more precision while lower threshold results in more recall.

Figure 6.13 shows the evaluation comparison of our proposed method with SIFT and BRISK baseline algorithms. For better visualization of the results, a heat map of the number of matches between each two

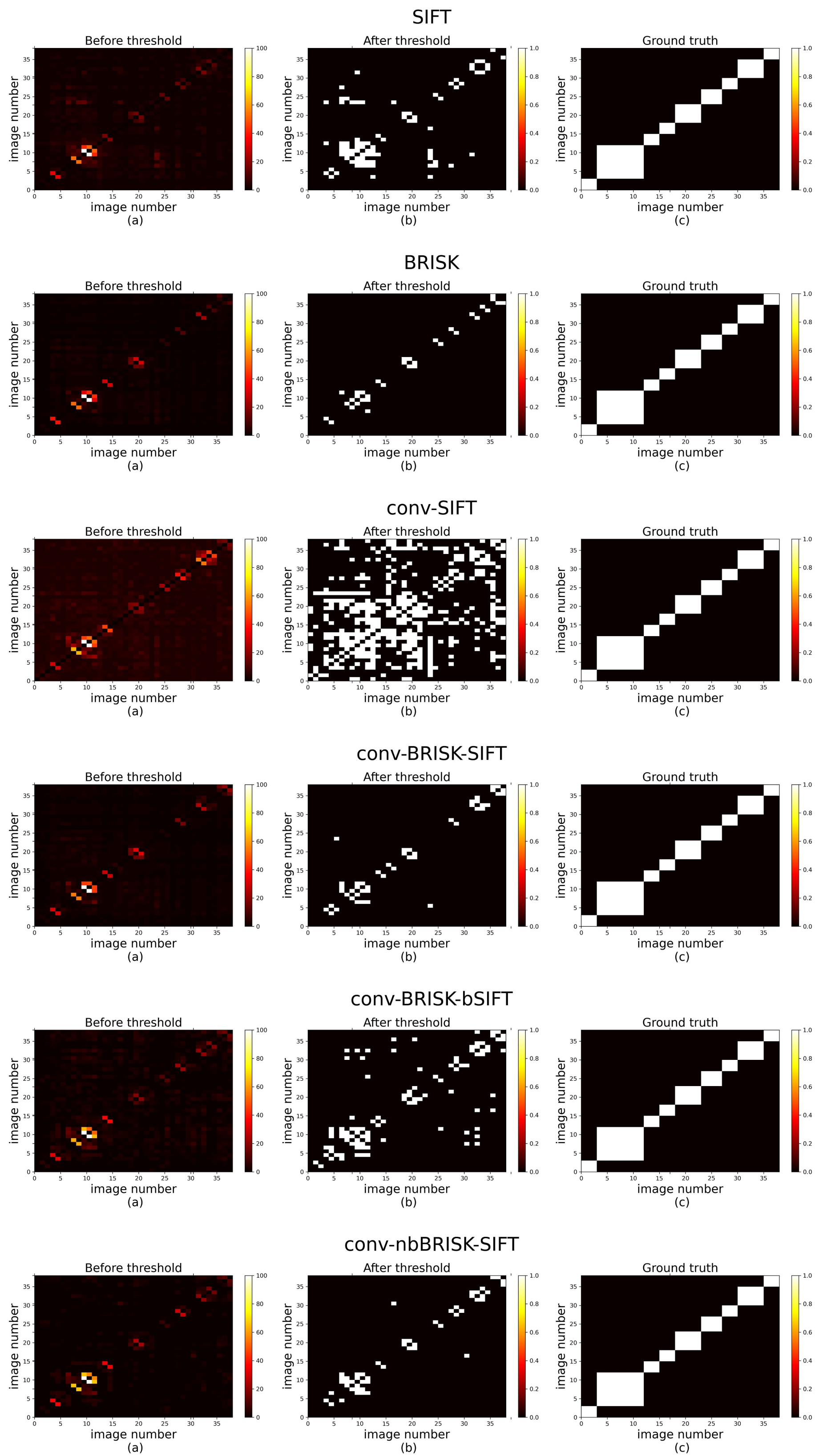

Figure 6.13: Heatmap of confusion matrix.

Table 6.4: Comparison of our methods with other work

Method	True positive	False positive	False Negative	Precision	Recall	F-score
AKAZE	68	8	124	0.89	0.35	0.25
SIFT	52	34	140	0.6	0.27	0.19
BRISK	34	0	158	1.0	0.18	0.15
ORB	148	952	44	0.13	0.77	0.11
conv-SIFT	102	298	90	0.26	0.53	0.17
conv-BRISK-SIFT	46	2	146	0.96	0.24	0.19
conv-BRISK-bSIFT	74	26	118	0.74	0.39	0.25
conv-nbBRISK-SIFT	50	2	142	0.96	0.26	0.2

images in the image sets is shown in Fig. 6.13. Column (a) in each row presents the normalized number of matches based on the maximum of that method. The figures in column (b) represent the decisions on the matches after applying a threshold on the column (a)'s values. Column (c) of each row in Fig. 6.13 shows the ground truth of the heat map. In the ground truth, each white pixel shows that the images corresponding to the vertical and horizontal image numbers are for the same set (same badgers). The black pixels (every pixel that is not white) in the ground truth represent the images being from different sets (not the same badgers). The performance of the algorithms can be visually compared in terms of the number of true positives and false positives as shown in Fig. 6.13. Conv-SIFT has the most false positives while BRISK shows the least number of false positives. Although conv-nbBRISK-SIFT and conv-BRISK-SIFT each shows two false positives, they have higher number of true positives than the BRISK algorithm. Figure 6.14 shows four examples of matches found between images of the same sets.

Table 6.4 shows the evaluation result of various commonly-used handcrafted methods on finding matches among badger faces based on the heat maps after applying the threshold shown in Fig. 6.13. The results of AKAZE [29], ORB [12], SIFT and BRISK in Table 6.4 are based on OpenCV implementation [165]. In Table 6.4, we present the precision, recall and F-score metrics. Precision shows the number of correctly selected badgers with respect to all proposed similar badgers while recall shows the number of correctly selected pairs over the number of all pairs of similar badgers in the dataset. If the purpose of an application is not missing any matchings among badger images, the algorithm with higher recall is preferable. However, if the number of correct pairs of badger images is important, the algorithm with higher precision has preference. The conv-SIFT method has lower precision but higher recall in comparison with SIFT and BRISK baselines which is also shown in the heat map in Fig. 6.13. The three methods that combine SIFT and BRISK demonstrate higher precision than SIFT and higher recall than BRISK. The BRISK algorithm shows the highest precision value while the recall value of BRISK is less than other methods. The AKAZE [29] algorithm has a similar F-score to conv-BRISK-bSIFT but requires more computations for non-linear scale-space generation. The ORB [12] algorithm has the highest recall value but the precision is less than other work.

The results indicate that although our badger identification dataset is challenging, our proposed methods (conv-BRISK-bSIFT and conv-nbBRISK-SIFT) attain higher F-score than the baseline algorithms. All of our proposed methods are trained only on the HPatches dataset and the weights of the convolutional filters were not fine-tuned on the badger data in the training stage. It is expected that better results can be obtained if the weights are trained using the badger data set.

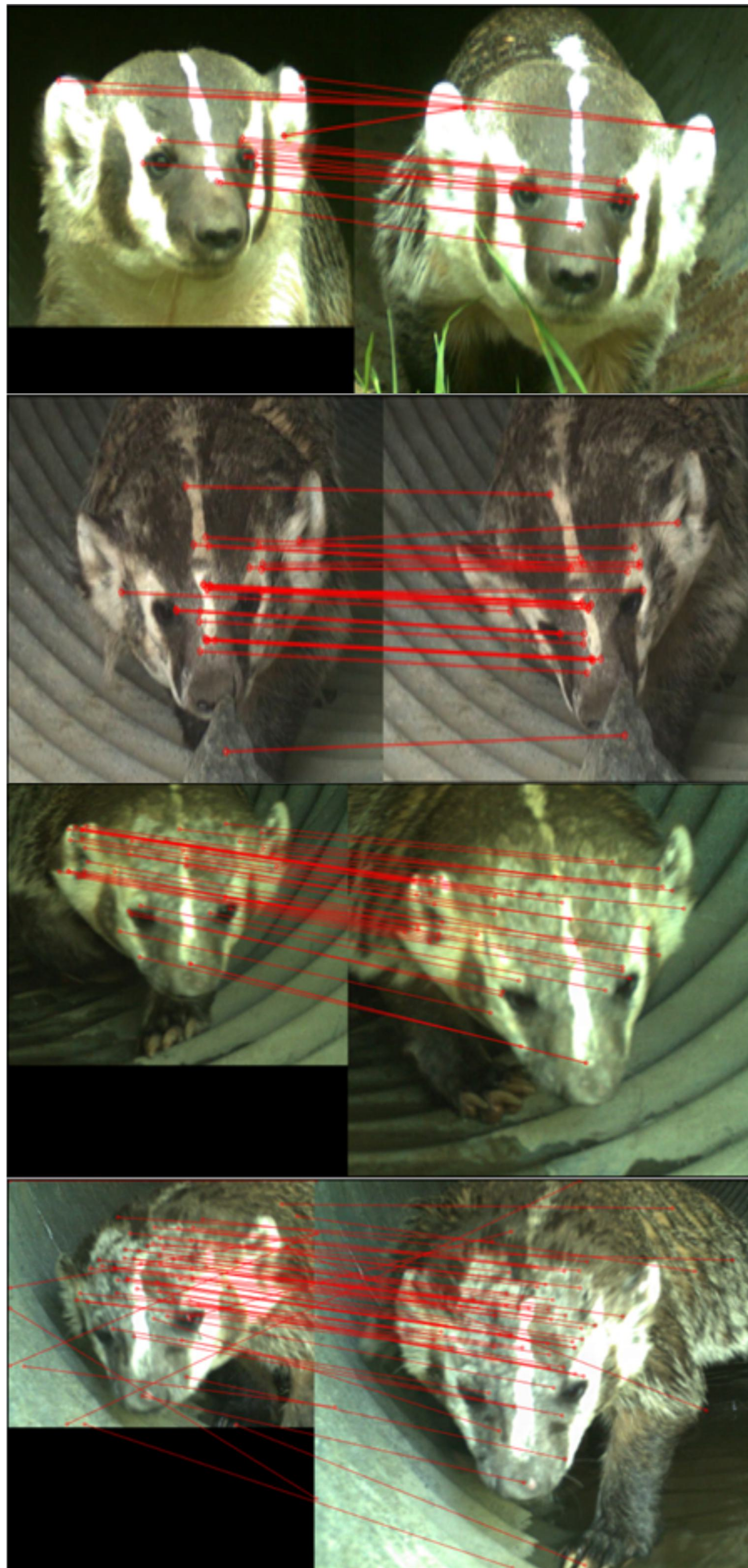

Figure 6.14: Examples of matches found on badger facial images.

6.6 Conclusion

In this work, we proposed a learning approach for convolutional prefiltering to improve image matching accuracy. The weights for the shallow convolutional filters are learned using a hill climbing algorithm. We also introduced a hardware accelerator design for prefiltering, keypoint detection, and description to increase the speed of image matching. We applied our method to the badger identification problem and demonstrated that our method outperforms the commonly-used SIFT, BRISK, and ORB descriptors.

A possible future path for this research is to conduct experiments with various kernel sizes of the convolutional neural network filters and analyze the size's effect on image matching accuracy. Another potential step is to gather a larger dataset of badgers. Using a higher number of images, we can fine-tune the weights of the CNN learning models to achieve higher accuracy.

Acknowledgment

This research was conducted based on the image dataset provided by the British Columbia Ministry of Transportation and Infrastructure.

Chapter 7

Conclusions

This chapter presents a summary of contributions of this dissertation and a discussion on possible next steps of this research. The outcomes and future paths for continuing this research are presented in sections 7.1 and 7.2, respectively.

7.1 Contributions

The focus of this dissertation is on improving image matching algorithms in terms of speed and accuracy. Deep neural networks have shown outstanding results in terms of accuracy in many computer vision applications. There have been many recent works using deep neural networks for image matching and the state-of-the-art in terms of accuracy is deep learning based method [31]. However, these methods require a high number of computations. Deep learning based methods are only comparable with handcrafted and non-deep learning based methods in terms of speed if they are implemented on accelerated platforms such as GPUs [25]. In addition, deep learning methods are dependent on the dataset used for training while the handcrafted methods are more generic. As a result, handcrafted image matching algorithms are still widely popular due to their efficiency and generality.

We decided to focus on hand crafted algorithms for image matching instead of deep learning methods to study solutions with higher speed and less computational requirements. At each chapter of this dissertation, we studied and presented methods to either increase the speed (chapters 2, 3, and 4) in comparison with other handcrafted methods, or increase the accuracy using methods with negligible latency increase (chapters 5 and 6) in comparison with deep learning methods. In other words, the objective was to work on the research gap between handcrafted and deep learning methods in this dissertation. Our contributions in this work can be applied to a wide variety of applications and can increase the speed of many computer vision systems which include image matching as the first step.

One of the popular methods to accelerate the execution of computer vision algorithms is an implementation on hardware platforms such as FPGAs. In chapter 2, we presented a comprehensive survey and analysis on FPGA-based implementations of the HOG algorithm. The HOG algorithm is a dense, non-binary descriptor with many applications. In addition to an analysis of the work in the literature, we proposed three design guidelines for HOG implementations to achieve higher speed, higher accuracy, or lower power consumption. In chapter 3, we introduced a novel hardware-software co-design of the HOG algorithm. Our

design in chapter 3 assigns the most computationally expensive parts of the HOG algorithm to the hardware platforms and the tasks that are expensive to implement on an FPGA to the software. This trade-off leads to an efficient, novel design in comparison with previous work in the literature.

We proposed a logarithm-based bin assignment to transform the required division in gradient orientation computation to subtraction. Implementation of a subtraction module is more efficient than a divider circuit on hardware platforms in terms of resource utilization. Our other contributions include parallel computation and a time-sharing protocol for histogram generation which results in the processing of one pixel per clock cycle for HOG descriptor generation. We also proposed a simplified block normalization logic to reduce hardware resource usage while maintaining accuracy. Our hardware-software co-design presented in chapter 3 attains a frame rate of 115 frames per second on a Xilinx® Kintex® Ultrascale™ FPGA while using fewer hardware resources, and only losing accuracy marginally, in comparison with existing work.

In chapter 4, a multi-scale FPGA-based implementation of the BRISK descriptor algorithm is proposed. A new sampling pattern for the BRISK algorithm with a similar number of sampling points to the original BRISK algorithm is presented, which reduces the number of registers for line-buffers by more than 50% and of the pipeline registers up to 73%. The proposed design is fully pipelined and achieves a maximum operating frequency of 168MHz. For images with 1920×1080 resolution, the proposed implementation runs at a frame rate of 78 fps, which makes it suitable for many real-time applications.

The proposed design of the multi-scale BRISK can be used as a part of a larger vision processing system. The input image can be read from memory or received directly from a camera. Our design can be added to various keypoint detectors for various applications. The keypoint detector part can be implemented in hardware or software and it does not affect the efficiency of the descriptor part. In addition, the proposed design can store the descriptors in on-chip memory, off-chip memory, or output them using a streaming protocol, depending on the application. Although the focus of chapter 4 is on the BRISK algorithm, the idea of using a hardware aware sampling pattern that facilitates hardware implementation for multi-scale processing could be adapted to other binary descriptors as well.

In chapter 5, we present an analysis on the combination of handcrafted algorithms containing complementary information to improve image matching accuracy. We proposed and compared three learning-based approaches for fusion of the descriptor algorithms. The first approach (weighted-fusion) combines the descriptor algorithms based on their computed distance. The second (binary-fusion) and third (non-binary-fusion) approaches are based on the direct fusion of the descriptor vectors. Binary-fusion combines a binary descriptor with a non-binary descriptor which is converted to a binary vector by a threshold learned on a training dataset. Non-binary fusion combines a non-binary descriptor with a binary descriptor which is scaled using a learned parameter from the training dataset. The proposed approaches have minimal dataset dependency, and are computationally efficient in comparison with deep neural network descriptors. In addition, an experimental case study on BRISK and SIFT algorithms is provided which shows that our approaches can improve mean Average Precision as demonstrated by using cross-validation on the HPatches dataset [11]. The proposed fusion approaches increase the accuracy of image matching applications while the individual non-computationally expensive handcrafted algorithms used in fusion can be executed in parallel to prevent increasing the latency.

In chapter 6, we introduced convolutional prefiltering for image matching. Using this method, we filtered the images before the keypoint detection step in the image matching pipeline. The weights of the filters are learned on a training data set using a hill climbing algorithm in a way such that the overall image

matching accuracy increases. We also introduced a hardware architecture for combining SIFT and BRISK algorithms in addition to convolutional prefiltering. We applied our methods to a challenging wildlife (badger identification) problem which finds the same individual badgers from various images using image matching techniques. We demonstrated that our proposed methods improve accuracy with respect to the baseline algorithms.

7.2 Future work

In this section, possible future paths for continuing the research in this dissertation are discussed.

Analysis and comparison of FPGA-based SIFT implementations

The SIFT descriptor algorithm is a popular non-binary descriptor algorithm very similar to HOG, which extracts features from local patches. We surveyed the FPGA-based implementations of the HOG algorithm in chapter 2. However, there are many FPGA-based implementations of the SIFT algorithm in the literature as well. To the best of our knowledge, there is not a comprehensive analysis of various FPGA-based implementations of the SIFT algorithm. One possible avenue for future research would be to survey and analyze hardware implementations of SIFT and other handcrafted descriptor algorithms, similar to our work for the HOG algorithm as presented in chapter 2.

Improvements on HOG hardware

Our hardware-software co-design presented in chapter 3 has the capability to be used in a system containing several HOG-SVM IP-cores in parallel for one image. One way to improve our current design in terms of speed is to consider parallelism of the HOG-SVM core. This modification requires designing a control unit to manage individual cores to 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 time to perform other tasks while the HOG-SVM cores and DMAs are processing the image. Another possible enhancement is to develop other variants of the HOG algorithm and their implementation on hardware. There are many other variants of the HOG algorithm such as HOG-3D [93] which require a high number of computations and could benefit from parallel implementation.

Improving the BRISK Hardware implementation

There are multiple potential enhancements that can be addressed in the next steps of research for our proposed design on the BRISK algorithm. The design presented in chapter 4 uses clock gating as a technique to reduce power consumption. The initial idea is to stop the clock signal for registers involved in description stages. An improvement to this design is to implement a gated filter unit directly after the keypoint detection as the next stage of the pipeline to reduce power consumption. This solution requires modification on the control unit structure to accommodate the required delays as well.

There are many keypoint detection algorithms presented in the literature [8], [19], [122]. In chapter 4, we use the FAST detector due to our focus on increasing the speed of the matching system. Studying other more complex detector algorithms to replace the FAST detector can lead to a design with increased accuracy.

Since keypoint detection is an early stage of the image matching pipeline, choosing an appropriate detector could improve accuracy.

In chapter 4, we assumed that the streaming input image enters our architecture at each clock cycle. This can be a valid assumption in many implementations and applications. However, for other applications the image may have been already loaded in on-chip memory. Therefore, a study on non-streaming input architectures for the BRISK algorithm is a potential future path for this research.

Combination and fusion of descriptor algorithms

Although in this dissertation we focused on analyzing the combination of a handcrafted non-binary descriptor algorithm with a handcrafted binary descriptor algorithm to prevent computationally expensive operations, one future path of this research is to study the combination of deep learning methods using the provided approaches and analyze the improvement on the accuracy of the state of the art methods. Another future step can be a comprehensive study and experiment on other handcrafted binary and non-binary algorithm test cases for fusion. Comprehensive experimental results on multiple pairs of various algorithms can be a basis for further analysis of descriptor algorithms based on their complementary information.

Fusion of complementary descriptors using neural networks

Our work in chapter 5 focuses on using simple methods such as conversion to binary or non-binary for fusion of the descriptor algorithms. To extend this research, combining descriptor vectors using other machine learning algorithms can be studied as well. One possible solution is fusion using shallow layers of a fully connected neural network on top of the extracted descriptor vectors using handcrafted algorithms. We performed experiments for this combination with a neural network that has a small number of fully-connected layers. However, we did not achieve successful and noticeable performance improvement. Further analysis of this method by optimizing the parameters of the network, increasing the number of training data, and designing a loss function appropriate for fusion could be the next step of this research.

Extending the contributions on prefiltering images for image matching using convolutional filters

In chapter 6, we introduced prefiltering of images using a shallow convolutional neural network trained on a data set for improving image matching accuracy. We conducted experiments with various numbers of layers and sub layers for filtering the images. There are several options to further extend these experiments and analysis of our proposed methods in chapter 6. One option is to design experiments to study the effect of various sizes of convolutional filters using our approach. Using various filter sizes can directly affect the smoothing of the pixels and the performance of detecting the keypoints. We used random weight values for initialization of the convolutional kernels for prefiltering. Another option would be to use commonly-known filters such as Sobel or Gaussian filters as the initial weights of the convolutional layers and fine-tune the weights based on the training data set, and analyze the effect of using prelearned fine-tuned filters for image matching. This approach leads to utilizing the advantages of traditional handcrafted image filters to more recent approaches.

References

[1] M. Agrawal, K. Konolige, and M. R. Blas, "CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching," in *Computer Vision – ECCV 2008*, D. Forsyth, P. Torr, and A. Zisserman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 102–115.

[2] C. Cadena, L. Carlone, H. Carrillo, *et al.*, "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016. DOI: 10.1109/TRO.2016.2624754.

[3] Y.-m. Wei, L. Kang, B. Yang, and L.-d. Wu, "Applications of structure from motion: a survey," *Journal of Zhejiang University SCIENCE C*, vol. 14, no. 7, pp. 486–494, Jul. 2013, ISSN: 1869-196X. DOI: 10.1631/jzus.CIDE1302. [Online]. Available: https://doi.org/10.1631/jzus.CIDE1302.

[4] 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. DOI: 10.1007/s11263-020-01359-2. [Online]. Available: https://doi.org/10.1007/s11263-020-01359-2.

[5] B. Fan, H. Liu, H. Zeng, J. Zhang, X. Liu, and J. Han, "Deep Unsupervised Binary Descriptor Learning Through Locality Consistency and Self Distinctiveness," eng, *IEEE Transactions on Multimedia*, vol. 23, pp. 2770–2781, 2021, ISSN: 1520-9210.

[6] X. Wang, X. Zeng, Y. Lyu, K. Chen, Y. Zhang, and D. Li, "A Deep Quadruplet Network for Local Descriptor Learning," *IEEE Access*, vol. 8, pp. 16 807–16 815, 2020. DOI: 10.1109/ACCESS.2019.2962624.

[7] Y. Dong, D. Fan, Q. Ma, and S. Ji, "Superpixel-Based Local Features for Image Matching," *IEEE Access*, vol. 9, pp. 15 467–15 484, 2021. DOI: 10.1109/ACCESS.2021.3052502.

[8] C. Harris and M. Stephens, "A combined corner and edge detector," in *In Proc. of Fourth Alvey Vision Conference*, 1988, pp. 147–151.

[9] K. Mikolajczyk and C. Schmid, "An affine invariant interest point detector," in *Computer Vision — ECCV 2002*, A. Heyden, G. Sparr, M. Nielsen, and P. Johansen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 128–142, ISBN: 978-3-540-47969-7.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Y. Tian, B. Fan, and F. Wu, "L2-Net: Deep Learning of Discriminative Patch Descriptor in Euclidean Space," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6128–6136. DOI: 10.1109/CVPR.2017.649.

[19] 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.

[20] 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.

[21] A. Alahi, R. Ortiz, and P. Vandergheynst, "FREAK: Fast Retina Keypoint," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 510–517. DOI: 10.1109/CVPR.2012.6247715.

[22] 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.

[23] D. Bekele, M. Teutsch, and T. Schuchert, "Evaluation of binary keypoint descriptors," in *2013 IEEE International Conference on Image Processing*, 2013, pp. 3652–3656. DOI: 10.1109/ICIP.2013.6738753.

[24] A. Mishchuk, D. Mishkin, F. Radenovic, and J. Matas, "Working hard to know your neighbor's margins: Local descriptor learning loss," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, *et al.*, Eds., 2017, pp. 4826–4837. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/831caa1b600f852b7844499430ecac17-Abstract.html.

[25] F. Bellavia and C. Colombo, "Is There Anything New to Say About SIFT Matching?" *International Journal of Computer Vision*, Mar. 2020. DOI: 10.1007/s11263-020-01297-z.

[26] I. Suárez, G. Sfeir, J. M. Buenaposada, and L. Baumela, "BEBLID: Boosted efficient binary local image descriptor," *Pattern Recognition Letters*, vol. 133, pp. 366–372, 2020, ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2020.04.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167865520301252.

[27] T. Trzcinski, M. Christoudias, P. Fua, and V. Lepetit, "Boosting Binary Keypoint Descriptors," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 2874–2881. DOI: 10.1109/CVPR.2013.370.

[28] Y. Gao, W. Huang, and Y. Qiao, "Learning multiple local binary descriptors for image matching," *Neurocomputing*, vol. 266, pp. 239–246, 2017, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.05.038. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231217308755.

[29] 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.

[30] I. Suárez, G. Sfeir, J. M. Buenaposada, and L. Baumela, "BELID: Boosted Efficient Local Image Descriptor," in *Pattern Recognition and Image Analysis*, A. Morales, J. Fierrez, J. S. Sánchez, and B. Ribeiro, Eds., Cham: Springer International Publishing, 2019, pp. 449–460, ISBN: 978-3-030-31332-6.

[31] H. Pan, Y. Chen, Z. He, F. Meng, and N. Fan, "TCDesc: Learning Topology Consistent Descriptors for Image Matching," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 32, no. 5, pp. 2845–2855, 2022. DOI: 10.1109/TCSVT.2021.3099846.

[32] I. D. Mienye and Y. Sun, "A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects," *IEEE Access*, vol. 10, pp. 99 129–99 149, 2022. DOI: 10.1109/ACCESS.2022.3207287.

[33] 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.

[34] 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.

[35] E. Azimi, A. Behrad, M. B. Ghaznavi-Ghoushchi, and J. Shanbehzadeh, "A fully pipelined and parallel hardware architecture for real-time BRISK salient point extraction," *Journal of Real-Time Image Processing*, vol. 16, no. 5, pp. 1859–1879, Oct. 2019, ISSN: 1861-8219. DOI: 10.1007/s11554-017-0693-4. [Online]. Available: https://doi.org/10.1007/s11554-017-0693-4.

[36] O. Ulusel, C. Picardo, C. B. Harris, S. Reda, and R. I. Bahar, "Hardware acceleration of feature detection and description algorithms on low-power embedded platforms," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577310.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] S. Ghaffari, K. F. Li, and D. W. Capson, "Learning-based fusion of local descriptors for improving image matching accuracy," to be submitted.

[42] S. Ghaffari, D. W. Capson, and K. F. Li, "Improving handcrafted image matching accuracy using shallow convolutional neural network filters and hardware acceleration," To be submitted.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] S. Advani, Y. Tanabe, K. Irick, J. Sampson, and V. Narayanan, "A scalable architecture for multi-class visual object detection," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015, pp. 1–8. DOI: 10.1109/FPL.2015.7293961.

[51] T. Sledevič, A. Serackis, and D. Plonis, "FPGA-Based Selected Object Tracking Using LBP, HOG and Motion Detection," in *2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, 2018, pp. 1–5. DOI: 10.1109/AIEEE.2018.8592410.

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

[53] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "FPGA-Based Real-Time Pedestrian Detection on High-Resolution Images," eng, in *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, IEEE, 2013, pp. 629–635, ISBN: 9780769549903.

[54] X. Ma, W. Najjar, and A. Roy-Chowdhury, "High-Throughput Fixed-Point Object Detection on FPGAs," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 107–107. DOI: 10.1109/FCCM.2014.40.

[55] M. Hemmati, M. Biglari-Abhari, S. Berber, and S. Niar, "HOG Feature Extractor Hardware Accelerator for Real-Time Pedestrian Detection," in *2014 17th Euromicro Conference on Digital System Design*, 2014, pp. 543–550. DOI: 10.1109/DSD.2014.60.

[56] X. Yuan, L. Cai-nian, X. Xiao-liang, J. Mei, and Z. Jian-guo, "A two-stage hog feature extraction processor embedded with SVM for pedestrian detection," in *2015 IEEE International Conference on Image Processing (ICIP)*, 2015, pp. 3452–3455. DOI: 10.1109/ICIP.2015.7351445.

[57] 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.

[58] Y. Nishizumi, G. Matsukawa, K. Kajihara, *et al.*, "FPGA implementation of object recognition processor for HDTV resolution video using sparse FIND feature," in *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, 2017, pp. 1–6. DOI: 10.1109/SiPS.2017.8109993.

[59] 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.

[60] 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.

[61] B. Meus, T. Kryjak, and M. Gorgon, "Embedded vision system for pedestrian detection based on HOG+SVM and use of motion information implemented in Zynq heterogeneous device," in *2017 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, 2017, pp. 406–411. DOI: 10.23919/SPA.2017.8166901.

[62] M.-S. Wang and Z.-R. Zhang, "FPGA implementation of HOG based multi-scale pedestrian detection," in *2018 IEEE International Conference on Applied System Invention (ICASI)*, 2018, pp. 1099–1102. DOI: 10.1109/ICASI.2018.8394472.

[63] 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.

[64] 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.

[65] 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.

[66] T. Wilson, M. Glatz, and M. Hodlmoser, "Pedestrian detection implemented on a fixed-point parallel architecture," in *2009 IEEE 13th International Symposium on Consumer Electronics*, 2009, pp. 47–51. DOI: 10.1109/ISCE.2009.5156970.

[67] S. Bauer, S. Köhler, K. Doll, and U. Brunsmann, "FPGA-GPU architecture for kernel SVM pedestrian detection," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, 2010, pp. 61–68. DOI: 10.1109/CVPRW.2010.5543772.

[68] C. Blair, N. M. Robertson, and D. Hume, "Characterizing a Heterogeneous System for Person Detection in Video Using Histograms of Oriented Gradients: Power Versus Speed Versus Accuracy," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 2, pp. 236–247, 2013. DOI: 10.1109/JETCAS.2013.2256821.

[69] P.-Y. Chen, C.-C. Huang, C.-Y. Lien, and Y.-H. Tsai, "An Efficient Hardware Implementation of HOG Feature Extraction for Human Detection," eng, *IEEE transactions on intelligent transportation systems*, vol. 15, no. 2, pp. 656–662, 2014, ISSN: 1524-9050.

[70] P.-Y. Hsiao, S.-Y. Lin, and S.-S. Huang, "An FPGA based human detection system with embedded platform," eng, *Microelectronic engineering*, vol. 138, pp. 42–46, 2015, ISSN: 0167-9317.

[71] 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.

[72] T. Adiono, K. S. Prakoso, C. Deo Putratama, B. Yuwono, and S. Fuada, "Practical Implementation of A Real-time Human Detection with HOG-AdaBoost in FPGA," in *TENCON 2018 - 2018 IEEE Region 10 Conference*, 2018, pp. 0211–0214. DOI: 10.1109/TENCON.2018.8650453.

[73] P. Ranawaka, M. Ekpanyapong, A. Tavares, J. Cabral, K. Athikulwongse, and V. Silva, "Application Specific Architecture for Hardware Accelerating HOG-SVM to Achieve High Throughput on HD Frames," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160-052X, 2019, pp. 131–134. DOI: 10.1109/ASAP.2019.00-18.

[74] M.-E. Ilas, "HOG algorithm simplification and its impact on FPGA implementation: With applications in car detection," in *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2017, pp. 1–6. DOI: 10.1109/ECAI.2017.8166450.

[75] 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.

[76] M.-E. Ilas, "Improved binary HOG algorithm and possible applications in car detection," in *2017 IEEE 23rd International Symposium for Design and Technology in Electronic Packaging (SIITME)*, 2017, pp. 274–279. DOI: 10.1109/SIITME.2017.8259907.

[77] Y. Zhou, Z. Chen, and X. Huang, "A pipeline architecture for traffic sign classification on an FPGA," eng, in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2015, pp. 950–953, ISBN: 9781479983919.

[78] 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.

[79] 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.

[80] X. Chen, J. Xu, and Z. Yu, "A fast and energy efficient FPGA-based system for real-time object tracking," in *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2017, pp. 965–968. DOI: 10.1109/APSIPA.2017.8282162.

[81] 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.

[82] A. B.K., V. Venkatraman, A. R. Kumar, and S. D. S., "Accelerating real-time computer vision applications using HW/SW co-design," in *2017 International Conference on Computer, Communications and Electronics (Comptelix)*, 2017, pp. 458–463. DOI: 10.1109/COMPTELIX.2017.8004013.

[83] D. D. Gajski, *Principles of Digital Design*. USA: Prentice-Hall, Inc., 1996, ISBN: 0133011445.

[84] I. Ahmad, Z. Ul Islam, F. Ullah, M. Abbas Hussain, and S. Nabi, "An FPGA Based Approach For People Counting Using Image Processing Techniques," in *2019 11th International Conference on Knowledge and Smart Technology (KST)*, 2019, pp. 148–152. DOI: 10.1109/KST.2019.8687568.

[85] E. P. R. Raj, B. S. Paul, and G. L. Narayanan, "Simplified SIFT Histogram of Oriented Gradients Bin Locator on FPGA," in *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2018, pp. 1–4. DOI: 10.1109/ICCCNT.2018.8493928.

[86] Priyanka and D. Kumar, "Feature Extraction and Selection of kidney Ultrasound Images Using GLCM and PCA," *Procedia Computer Science*, vol. 167, pp. 1722–1731, 2020, International Conference on Computational Intelligence and Data Science, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.03.382. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920308486.

[87] M. A. Djeziri, S. Benmoussa, and E. Zio, "Review on Health Indices Extraction and Trend Modeling for Remaining Useful Life Estimatio," in *Artificial Intelligence Techniques for a Scalable Energy Transition: Advanced Methods, Digital Technologies, Decision Support Tools, and Applications*, M. Sayed-Mouchaweh, Ed. Cham: Springer International Publishing, 2020, pp. 183–223, ISBN: 978-3-030-42726-9. DOI: 10.1007/978-3-030-42726-9_8. [Online]. Available: https://doi.org/10.1007/978-3-030-42726-9_8.

[88] V. Montalvo, A. A. Estévez-Bén, J. Rodríguez-Reséndiz, G. Macias-Bobadilla, J. D. Mendiola-Santíbañez, and K. A. Camarillo-Gómez, "FPGA-Based Architecture for Sensing Power Consumption on Parabolic and Trapezoidal Motion Profiles," *Electronics*, vol. 9, no. 8, 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9081301. [Online]. Available: https://www.mdpi.com/2079-9292/9/8/1301.

[89] X. Zhang, X. Wei, Q. Sang, H. Chen, and Y. Xie, "An Efficient FPGA-Based Implementation for Quantized Remote Sensing Image Scene Classification Network," *Electronics*, vol. 9, no. 9, 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9091344. [Online]. Available: https://www.mdpi.com/2079-9292/9/9/1344.

[90] M. Zhao, C. Hu, F. Wei, K. Wang, C. Wang, and Y. Jiang, "Real-Time Underwater Image Recognition with FPGA Embedded System for Convolutional Neural Network," *Sensors*, vol. 19, no. 2, 2019, ISSN: 1424-8220. DOI: 10.3390/s19020350. [Online]. Available: https://www.mdpi.com/1424-8220/19/2/350.

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

[92] *INRIA Person dataset*, 2005. [Online]. Available: http://pascal.inrialpes.fr/data/human/.

[93] 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.

[94] E. Mair, G. D. Hager, D. Burschka, M. Suppa, and G. Hirzinger, "Adaptive and Generic Corner Detection Based on the Accelerated Segment Test," in *Computer Vision – ECCV 2010*, K. Daniilidis, P. Maragos, and N. Paragios, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 183–196.

[95] R. Liu, J. Yang, Y. Chen, and W. Zhao, “eSLAM: An Energy-Efficient Accelerator for Real-Time ORB-SLAM on FPGA Platform*,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[96] M. Kashif, T. M. Deserno, D. Haak, and S. Jonas, “Feature description with SIFT, SURF, BRIEF, BRISK, or FREAK? A general question answered for bone age assessment,” *Computers in Biology and Medicine*, vol. 68, pp. 67–75, 2016, ISSN: 0010-4825.

[97] C. Belloni, N. Aouf, J. L. Caillec, and T. Merlet, “Comparison of Descriptors for SAR ATR,” in *2019 IEEE Radar Conference (RadarConf)*, 2019, pp. 1–6. DOI: 10.1109/RADAR.2019.8835804.

[98] 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.

[99] L. Zhang, K. Mistry, M. Jiang, S. Chin Neoh, and M. A. Hossain, “Adaptive facial point detection and emotion recognition for a humanoid robot,” *Computer Vision and Image Understanding*, vol. 140, pp. 93–114, 2015, ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2015.07.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1077314215001605.

[100] T. Mouats, N. Aouf, D. Nam, and S. Vidas, “Performance Evaluation of Feature Detectors and Descriptors Beyond the Visible,” *Journal of Intelligent and Robotic Systems*, vol. 92, no. 1, pp. 33–63, 2018. DOI: 10.1007/s10846-017-0762-8.

[101] S. Madeo and M. Bober, “Fast, Compact, and Discriminative: Evaluation of Binary Descriptors for Mobile Applications,” *IEEE Transactions on Multimedia*, vol. 19, no. 2, pp. 221–235, 2017. DOI: 10.1109/TMM.2016.2615521.

[102] R. Sun, P. Liu, J. Wang, C. Accetti, and A. A. Naqvi, “A 42fps full-HD ORB feature extraction accelerator with reduced memory overhead,” in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 183–190. DOI: 10.1109/FPT.2017.8280137.

[103] R. de Lima, J. Martinez-Carranza, A. Morales-Reyes, and R. Cumplido, “Improving the construction of ORB through FPGA-based acceleration,” *Machine Vision and Applications*, vol. 28, no. 5, pp. 525–537, Aug. 2017, ISSN: 1432-1769. DOI: 10.1007/s00138-017-0851-5. [Online]. Available: https://doi.org/10.1007/s00138-017-0851-5.

[104] P. Tran, T. H. Pham, S. K. Lam, M. Wu, and B. A. Jasani, “Stream-Based ORB Feature Extractor with Dynamic Power Optimization,” in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 94–101. DOI: 10.1109/FPT.2018.00024.

[105] W. Fang, Y. Zhang, B. Yu, and S. Liu, “FPGA-based ORB feature extraction for real-time visual SLAM,” in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 275–278. DOI: 10.1109/FPT.2017.8280159.

[106] L. Kalms, M. Hajduk, and D. Göhringer, “Efficient Pattern Recognition Algorithm Including a Fast Retina Keypoint FPGA Implementation,” in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 121–128. DOI: 10.1109/FPL.2019.00028.

[107] R. Kapela, K. Gugala, P. Sniatala, A. Swietlicka, and K. Kolanowski, “Embedded platform for local image descriptor based object detection,” *Applied Mathematics and Computation*, vol. 267, pp. 419–426, 2015, The Fourth European Seminar on Computing (ESCO 2014), ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2015.02.029. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S009630031500209X.

[108] T. H. Pham, P. Tran, and S. Lam, “High-Throughput and Area-Optimized Architecture for rBRIEF Feature Extraction,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 4, pp. 747–756, 2019. DOI: 10.1109/TVLSI.2018.2881105.

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

[110] 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.

[111] 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.

[112] Y. Rao, J. Yang, Y. Ju, *et al.*, “Learning General Feature Descriptor for Visual Measurement With Hierarchical View Consistency,” *IEEE Transactions on Instrumentation and Measurement*, vol. 71, pp. 1–12, 2022. DOI: 10.1109/TIM.2022.3169563.

[113] F. Wang, Z. Liu, H. Zhu, P. Wu, and C. Li, “An Improved Method for Stable Feature Points Selection in Structure-from-Motion Considering Image Semantic and Structural Characteristics,” *Sensors*, vol. 21, no. 7, 2021, ISSN: 1424-8220. DOI: 10.3390/s21072416. [Online]. Available: https://www.mdpi.com/1424-8220/21/7/2416.

[114] L. Cai, Y. Ye, X. Gao, Z. Li, and C. Zhang, “An improved visual SLAM based on affine transformation for ORB feature extraction,” *Optik*, vol. 227, p. 165 421, 2021, ISSN: 0030-4026. DOI: https://doi.org/10.1016/j.ijleo.2020.165421. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0030402620312572.

[115] M. Cao, H. Gao, and W. Jia, “Stable image matching for 3D reconstruction in outdoor,” eng, *International journal of circuit theory and applications*, vol. 49, no. 7, pp. 2274–2289, 2021, ISSN: 0098-9886.

[116] M. Bansal, M. Kumar, and M. Kumar, “2D object recognition: a comparative analysis of SIFT, SURF and ORB feature descriptors,” eng, *Multimedia tools and applications*, vol. 80, no. 12, pp. 18 839–18 857, 2021, ISSN: 1380-7501.

[117] S. Shirmohammadi and A. Ferrero, “Camera as the instrument: The rising trend of vision based measurement,” *IEEE Instrumentation & Measurement Magazine*, vol. 17, no. 3, pp. 41–47, 2014. DOI: 10.1109/MIM.2014.6825388.

[118] H. Zhang, Z. Tang, Y. Xie, and W. Gui, “RPI-SURF: A Feature Descriptor for Bubble Velocity Measurement in Froth Flotation With Relative Position Information,” *IEEE Transactions on Instrumentation and Measurement*, vol. 70, pp. 1–14, 2021. DOI: 10.1109/TIM.2021.3102738.

[119] J. Yuan, S. Zhu, K. Tang, and Q. Sun, “ORB-TEDM: An RGB-D SLAM Approach Fusing ORB Triangulation Estimates and Depth Measurements,” *IEEE Transactions on Instrumentation and Measurement*, vol. 71, pp. 1–15, 2022. DOI: `10.1109/TIM.2022.3154800`.

[120] C. Wang, R. Xu, S. Xu, W. Meng, and X. Zhang, “CNDesc: Cross Normalization for Local Descriptors Learning,” *IEEE Transactions on Multimedia*, pp. 1–1, 2022. DOI: `10.1109/TMM.2022.3169331`.

[121] M. Dusmanu, I. Rocco, T. Pajdla, *et al.*, “D2-Net: A Trainable CNN for Joint Description and Detection of Local Features,” eng, in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2019, pp. 8084–8093, ISBN: 9781728132938.

[122] D. DeTone, T. Malisiewicz, and A. Rabinovich, “SuperPoint: Self-Supervised Interest Point Detection and Description,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2018, pp. 337–33 712. DOI: `10.1109/CVPRW.2018.00060`.

[123] X. Zhao, X. Wu, J. Miao, W. Chen, P. C. Y. Chen, and Z. Li, “ALIKE: Accurate and Lightweight Keypoint Detection and Descriptor Extraction,” eng, *IEEE transactions on multimedia*, pp. 1–1, 2022, ISSN: 1520-9210.

[124] U. Efe, K. G. Ince, and A. A. Alatan, “Effect of Parameter Optimization on Classical and Learning-based Image Matching Methods,” eng, 2021.

[125] Y. Jin, D. Mishkin, A. Mishchuk, *et al.*, “Image Matching Across Wide Baselines: From Paper to Practice,” eng, *International journal of computer vision*, vol. 129, no. 2, pp. 517–547, 2021, ISSN: 0920-5691.

[126] T. Imsaengsuk and S. Pumrin, “Feature Detection and Description based on ORB Algorithm for FPGA-based Image Processing,” eng, in *2021 9th International Electrical Engineering Congress (iEECON)*, IEEE, 2021, pp. 420–423, ISBN: 1728195845.

[127] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, “Hardware for machine learning: Challenges and opportunities,” eng, in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, Ithaca: IEEE, 2017, pp. 1–8, ISBN: 9781509051915.

[128] H. A. Alshazly, M. Hassaballah, A. A. Ali, and G. Wang, “An Experimental Evaluation of Binary Feature Descriptors,” eng, in *Proceedings of the International Conference on Advanced Intelligent Systems and Informatics 2017*, ser. Advances in Intelligent Systems and Computing, Cham: Springer International Publishing, 2017, pp. 181–191, ISBN: 3319648608.

[129] M. O. Salameh, A. Abdullah, and S. Sahran, “Ensemble of Vector and Binary Descriptor for Loop Closure Detection,” eng, in *Robot Intelligence Technology and Applications 4*, ser. Advances in Intelligent Systems and Computing, Cham: Springer International Publishing, 2016, pp. 329–340, ISBN: 331931291X.

[130] G. A. Siva Raja, M. Siddart, S. C. Kashyap, and P. Ramadevi, “Comprehensive Analysis of Fused Descriptors for Image Retrieval,” eng, in *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, vol. 1, Piscataway: IEEE, 2021, pp. 01–05, ISBN: 9781665400169.

[131] B. Fan, Q. Kong, X. Yuan, Z. Wang, and C. Pan, "Learning weighted Hamming distance for binary descriptors," eng, in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, IEEE, 2013, pp. 2395–2399, ISBN: 1479903566.

[132] B. Fan, Q. Kong, T. Trzcinski, Z. Wang, C. Pan, and P. Fua, "Receptive Fields Selection for Binary Feature Description," eng, *IEEE transactions on image processing*, vol. 23, no. 6, pp. 2583–2595, 2014, ISSN: 1057-7149.

[133] T. Trzcinski, M. Christoudias, and V. Lepetit, "Learning Image Descriptors with Boosting," eng, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 3, pp. 597–610, 2015, ISSN: 0162-8828.

[134] S. Husain and M. Bober, "On aggregation of local binary descriptors," eng, in *2016 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, IEEE, 2016, pp. 1–6, ISBN: 9781509015528.

[135] K. Iwamoto, R. Mase, and T. Nomura, "BRIGHT: A scalable and compact binary descriptor for low-latency and high accuracy object identification," eng, in *2013 IEEE International Conference on Image Processing*, IEEE, 2013, pp. 2915–2919, ISBN: 1479923419.

[136] T.-Y. Yang, J.-H. Hsu, Y.-Y. Lin, and Y.-Y. Chuang, "DeepCD: Learning Deep Complementary Descriptors for Patch Representations," eng, in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, 2017, pp. 3334–3342, ISBN: 9781538610329.

[137] H. Liu, Q. Zhang, B. Fan, Z. Wang, and J. Han, "Features Combined Binary Descriptor Based on Voted Ring-Sampling Pattern," eng, *IEEE transactions on circuits and systems for video technology*, vol. 30, no. 10, pp. 3675–3687, 2020, ISSN: 1051-8215.

[138] R. Arandjelović and A. Zisserman, "Three things everyone should know to improve object retrieval," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 2911–2918. DOI: 10.1109/CVPR.2012.6248018.

[139] G. Hua, M. Brown, and S. Winder, "Discriminant Embedding for Local Image Descriptors," eng, in *2007 IEEE 11th International Conference on Computer Vision*, IEEE, 2007, pp. 1–8, ISBN: 1424416302.

[140] Z. Dai, X. Huang, W. Chen, L. He, and H. Zhang, "A Comparison of CNN-Based and Hand-Crafted Keypoint Descriptors," eng, in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 2399–2404, ISBN: 9781538660270.

[141] T. S. Chathuranga and R. Munasinghe, "Aerial Image Matching Based Relative Localization of a UAV in Urban Environments," eng, in *2019 Moratuwa Engineering Research Conference (MERCon)*, IEEE, 2019, pp. 633–637, ISBN: 9781728136318.

[142] S. Schneider, G. W. Taylor, S. Linquist, S. C. Kremer, and R. B. O'Hara, "Past, present and future approaches using computer vision for animal re-identification from camera trap data," eng, *Methods in Ecology and Evolution*, vol. 10, no. 4, pp. 461–470, 2019, ISSN: 2041-210X.

[143] T. Petso, R. S. Jamisola, D. Mpoeleng, E. Bennitt, and W. Mmereki, "Automatic animal identification from drone camera based on point pattern analysis of herd behaviour," eng, *Ecological informatics*, vol. 66, pp. 101 485–, 2021, ISSN: 1574-9541.

[144] P. C. Ravoor and S. T.S.B., "Deep Learning Methods for Multi-Species Animal Re-identification and Tracking – a Survey," eng, *Computer science review*, vol. 38, pp. 100 289–, 2020, ISSN: 1574-0137.

[145] A. C. Ferreira, L. R. Silva, F. Renna, *et al.*, "Deep learning-based methods for individual recognition in small birds," eng, *Methods in ecology and evolution*, vol. 11, no. 9, pp. 1072–1085, 2020, ISSN: 2041-210X.

[146] E. Nepovinnykh, I. Chelak, A. Lushpanov, T. Eerola, H. Kälviäinen, and O. Chirkova, "Matching individual Ladoga ringed seals across short-term image sequences," eng, *Mammalian biology : Zeitschrift für Säugetierkunde*, vol. 102, no. 3, pp. 935–950, 2022, ISSN: 1616-5047.

[147] P. D. Alexander and D. J. Craighead, "A novel camera trapping method for individually identifying pumas by facial features," eng, *Ecology and evolution*, vol. 12, no. 1, e8536–n/a, 2022, ISSN: 2045-7758.

[148] M. Clapham, E. Miller, M. Nguyen, and R. C. Van Horn, "Multispecies facial detection for individual identification of wildlife: a case study across ursids," eng, *Mammalian biology : Zeitschrift für Säugetierkunde*, vol. 102, no. 3, pp. 921–933, 2022, ISSN: 1616-5047.

[149] N. Liu, Q. Zhao, N. Zhang, X. Cheng, and J. Zhu, "Pose-guided complementary features learning for amur tiger re-identification," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 286–293. DOI: 10.1109/ICCVW.2019.00038.

[150] D. Schofield, A. Nagrani, A. Zisserman, *et al.*, "Chimpanzee face recognition from videos in the wild using deep learning," eng, *Science advances*, vol. 5, no. 9, eaaw0736–eaaw0736, 2019, ISSN: 2375-2548.

[151] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects," eng, *IEEE transaction on neural networks and learning systems*, vol. PP, pp. 1–21, 2021, ISSN: 2162-237X.

[152] P. Kreowsky and B. Stabernack, "A Full-Featured FPGA-Based Pipelined Architecture for SIFT Extraction," eng, *IEEE access*, vol. 9, pp. 128 564–128 573, 2021, ISSN: 2169-3536.

[153] S.-A. Li, W.-Y. Wang, W.-Z. Pan, C.-C. J. Hsu, and C.-K. Lu, "FPGA-Based Hardware Design for Scale-Invariant Feature Transform," *IEEE Access*, vol. 6, pp. 43 850–43 864, 2018. DOI: 10.1109/ACCESS.2018.2863019.

[154] S. Chalup and F. Maire, "A study on hill climbing algorithms for neural network training," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 3, 1999, 2014–2021 Vol. 3. DOI: 10.1109/CEC.1999.785522.

[155] L. Hernando, A. Mendiburu, and J. A. Lozano, "Hill-Climbing Algorithm: Let's Go for a Walk Before Finding the Optimum," in *2018 IEEE Congress on Evolutionary Computation (CEC)*, 2018, pp. 1–7. DOI: 10.1109/CEC.2018.8477836.

[156] J. Vourvoulakis, J. Kalomiros, and J. Lygouras, "Fully pipelined FPGA-based architecture for real-time SIFT extraction," eng, *Microprocessors and microsystems*, vol. 40, pp. 53–73, 2016, ISSN: 0141-9331.

[157] G. Doménech-Asensi, J. Zapata-Pérez, R. Ruiz-Merino, *et al.*, "All-hardware SIFT implementation for real-time VGA images feature extraction," *Journal of Real-Time Image Processing*, vol. 17, no. 2, pp. 371–382, Apr. 2020, ISSN: 1861-8219. DOI: 10.1007/s11554-018-0781-0. [Online]. Available: https://doi.org/10.1007/s11554-018-0781-0.

[158] C.-H. Kuo, E.-H. Huang, C.-H. Chien, and C.-C. Hsu, "FPGA Design of Enhanced Scale-Invariant Feature Transform with Finite-Area Parallel Feature Matching for Stereo Vision," eng, *Electronics (Basel)*, vol. 10, no. 14, pp. 1632–, 2021, ISSN: 2079-9292.

[159] M. Clapham, E. Miller, M. Nguyen, and C. T. Darimont, "Automated facial recognition for wildlife that lack unique markings: A deep learning approach for brown bears," eng, *Ecology and evolution*, vol. 10, no. 23, pp. 12 883–12 892, 2020, ISSN: 2045-7758.

[160] Á. Villafañe-Trujillo, C. López-González, and J. Kolowski, "Throat Patch Variation in Tayra (Eira barbara) and the Potential for Individual Identification in the Field," eng, *Diversity (Basel)*, vol. 10, no. 1, pp. 7–, 2018, ISSN: 1424-2818.

[161] R. L. Harrison, "Noninvasive Identification of Individual American Badgers by Features of Their Dorsal Head Stripes," eng, *Western North American naturalist*, vol. 76, no. 2, pp. 259–261, 2016, ISSN: 1527-0904.

[162] *Badgers in BC*, May 2022. [Online]. Available: `https://badgers.bc.ca/faq/`.

[163] R. W. Klafki, "Road Ecology of a Northern Population of Badgers (TAXIDEA TAXUS) in British Columbia, Canada," Ph.D. dissertation, UNIVERSITY OF NORTHERN BRITISH COLUMBIA, 2014.

[164] M. J. Gould and R. L. Harrison, "A novel approach to estimating density of American badgers (Taxidea taxus) using automatic cameras at water sources in the Chihuahuan Desert," *Journal of Mammalogy*, vol. 99, no. 1, pp. 233–241, Nov. 2017, ISSN: 0022-2372. DOI: `10.1093/jmammal/gyx142`. eprint: `https://academic.oup.com/jmammal/article-pdf/99/1/233/23747888/gyx142.pdf`. [Online]. Available: `https://doi.org/10.1093/jmammal/gyx142`.

[165] *OpenCV: Feature Matching*, 2021. [Online]. Available: `https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html`.

Appendix A

Publications Arising from This Dissertation

- 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.
- 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.
- 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.
- 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.
- S. Ghaffari, K. F. Li, and D. W. Capson, "Learning-based fusion of local descriptors for improving image matching accuracy," to be submitted.
- S. Ghaffari, D. W. Capson, and K. F. Li, "Improving handcrafted image matching accuracy using shallow convolutional neural network filters and hardware acceleration," to be submitted.