



University
of Cyprus

**DEPARTMENT OF ELECTRICAL AND COMPUTER
ENGINEERING**

**REAL-TIME HARDWARE ACCELERATION OF
OBJECT DETECTION FOR INTELLIGENT
EMBEDDED VISION SYSTEMS**

DOCTOR OF PHILOSOPHY DISSERTATION

CHRISTOS KYRKOU

2014



**University
of Cyprus**

**DEPARTMENT OF ELECTRICAL AND COMPUTER
ENGINEERING**

**REAL-TIME HARDWARE ACCELERATION OF
OBJECT DETECTION FOR INTELLIGENT
EMBEDDED VISION SYSTEMS**

CHRISTOS KYRKOU

**A Dissertation Submitted to the University of Cyprus in Partial
Fulfillment of the Requirements of the Degree of Doctor of Philosophy**

May 2014

Christos Kyrkou

© CHRISTOS KYRKOU, 2014

VALIDATION PAGE

Doctoral Candidate: Christos Kyrkou

Doctoral Thesis Title: Real-Time Hardware Acceleration of Object Detection for Intelligent Embedded Vision Systems

*The present Doctorate Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the **Department of Electrical and Computer Engineering**, and was approved on January 15, 2014 by the members of the **Examination Committee**.*

Examination Committee:

Committee Chair



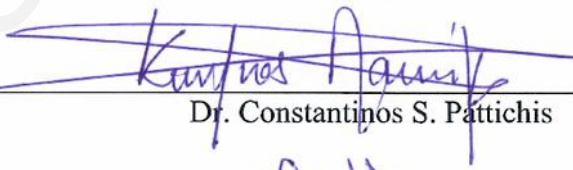
Dr. Marios Polycarpou

Research Supervisor



Dr. Theocharis Theocharides

Committee Member



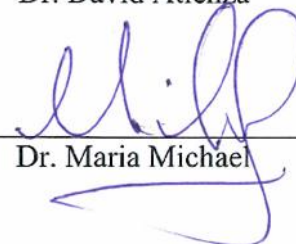
Dr. Constantinos S. Pattichis

Committee Member



Dr. David Atienza

Committee Member



Dr. Maria Michael

Christos Kyrkou

DECLARATION OF DOCTORAL CANDIDATE

The present doctoral dissertation was submitted in partial fulfillment of the requirements of the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

Christos Kyrkou

A handwritten signature in blue ink, appearing to read 'Christos Kyrkou', with a horizontal line drawn through it.

Christos Kyrkou

Christos Kyrkou

Περίληψη

Η ανίχνευση αντικειμένων είναι μια βασική και απαραίτητη λειτουργία για ενσωματωμένα συστήματα όρασης, καθώς τους δίνει τη δυνατότητα να αλληλεπιδρούν με πιο έξυπνο τρόπο με το περιβάλλον τους και γίνεται όλο και πιο αναγκαία για ένα ευρύ φάσμα εφαρμογών που σχετίζονται με την επεξεργασία εικόνων, ρομποτική, συστήματα ασφαλείας και βίο-πληροφορική. Υφιστάμενα συστήματα που βασίζονται μόνο σε υλοποίηση σε λογισμικό παρέχουν την απαιτούμενη απόδοση μόνο για εικόνες μικρών διαστάσεων και κάτω από ιδανικές συνθήκες. Συνεπώς, υπάρχει ανάγκη για ανάπτυξη αρχιτεκτονικών σε υλικό που μπορούν να χρησιμοποιηθούν σε διάφορες εφαρμογές για εύρεση διαφορετικών αντικειμένων, σε εικόνες διαφόρων μεγεθών.

Η διατριβή ασχολείται με μια βασική πρόκληση στις μέρες μας που σχετίζεται με το σχεδιασμό και την υλοποίηση αρχιτεκτονικών υλικού για την ανίχνευση αντικείμενων σε εικόνες, που να έχουν χαμηλή κατανάλωση ενέργειας και να λειτουργούν σε πραγματικό χρόνο, αξιοποιώντας τη χρήση αναδιατασσόμενου υλικού. Συγκεκριμένα, προτείνονται αρχιτεκτονικές για δύο αλγόριθμους, των αλγόριθμο Viola-Jones και τις διανυσματικές μηχανές υποστήριξης, με στόχο την ανίχνευση αντικειμένων σε πραγματικό χρόνο, διατηρώντας παράλληλα μια καλή ακρίβεια. Επιπλέον ένας ακόμη στόχος αυτής της διατριβής είναι η ανάπτυξη γενικών και επεκτάσιμων αρχιτεκτονικών υλικού, που να επιτρέπουν τη χρήση τους σε ένα ευρύ φάσμα εφαρμογών και διαφορετικές πλατφόρμες. Επιπλέον, ευφυείς μηχανισμοί ενσωματωθεί μαζί με τις προτεινόμενες αρχιτεκτονικές υλικού με σκοπό την περαιτέρω βελτίωση της απόδοσης. Επιπλέον, η διατριβή προτείνει μια μεθοδολογία για την περαιτέρω βελτίωση της απόδοσης των αλγορίθμων ανίχνευσης αντικείμενου με τη χρησιμοποίηση ακμών εικόνας, και τρισδιάστατων δεδομένων ώστε να επιτύχει την μείωση του αριθμού των παραθύρων που δημιουργούνται για μια εικόνα ώστε να αυξηθεί η συνολική απόδοση του συστήματος ανίχνευσης. Οι προτεινόμενες αρχιτεκτονικές αξιολογήθηκαν πειραματικά σε πλατφόρμες αναδιατασσόμενου υλικού και είναι ικανές να παρέχουν επεξεργαστικές επιδόσεις που ξεπερνούν τις 30 εικόνες ανά δευτερόλεπτο, επιτρέποντας να χρησιμοποιηθούν σε εφαρμογές ανίχνευσης αντικειμένων πραγματικού χρόνου.

Christos Kyrkou

Abstract

Object detection is a necessary task for embedded vision systems as it enables them to interact more intelligently with their host environment, and increases their responsiveness and awareness with regards to their surroundings. The typical object detection process involves extracting information of an image/video frame at various scales and classifying it using a machine learning pattern recognition algorithm to determine the presence of objects of interest. Object detection is a tedious process that needs to be performed within real-time constraints. Hence, this thesis addresses a key challenge nowadays related to the design and implementation of a hardware architecture for visual object detection in order to operate under low power requirements and facilitate real-time performance, both important constraints in embedded applications, while exploiting the use of reconfigurable hardware (FPGAs). In particular, two popular algorithms are implemented, the Viola-Jones and Support Vector Machines with the overall aim in increasing the performance of real-time object detection while maintaining the accuracy. Another goal of the thesis is to develop novel, generic and scalable architecture models in order to allow their usage in a wide range of applications and different hardware platforms. Furthermore, intelligent mechanisms are integrated along with the proposed hardware architecture in order to further improve performance or reduce computation overheads. In addition, and beyond the hardware implementation contributions, the thesis proposes a methodology to further improve the performance of object detection algorithms by utilizing edge image features, 3D depth data and classification in order to reduce the number of windows generated for an image and, hence, increase the overall performance of the detection problem. The proposed architectures are experimentally evaluated on Field Programmable Gate Arrays (FPGAs) computing platforms demonstrating real-time performance (over 30 frames-per second) that enables them to be used in real-time object detection applications, and maintain detection accuracy comparable to software implementations.

Keywords: *AdaBoost, Cascade Classifier, Disparity Estimation, Edge Detection, Field Programmable Gate Array (FPGA), Haar-features, Local Binary Pattern (LBP), Monolithic Classifier, Neural Network, Object Detection, Parallel Architecture, Real-time and Embedded Systems, Supervised Learning, Support Vector Machines (SVMs)*

Christos Kyrkou

Acknowledgements

My immense appreciation and wholehearted gratitude goes to my thesis and research supervisor Dr. Theocharis Theocharides for his guidance and support throughout the whole period of my studies, and for providing me with the opportunity to interact with leading researchers in the area of computer engineering. I also wish to express my warm and sincere thanks to PhD committee members from the University of Cyprus Dr. Marios Polycarpou, Dr. Maria K. Michael and Dr. Constantinos S. Pattichis for their feedback, comments, and insightful advice. I would like to extend a special thanks to Dr. David Atienza from École Polytechnique Federale de Lausanne (EPFL), who as an external member, provided an insightful perspective to the research in this thesis. I would also like to thank Dr. Christos-Savvas Bouganis, from Imperial London College for his collaboration and insights for part of the research done in this thesis.

I would also like to thank my colleagues at KIOS Research Center and fellow members of the Embedded and Application-specific System-on-Chip laboratory for their sincere friendship, and interesting discussions and experiences we shared together as part of the same research group.

Finally and most importantly I am deeply grateful to my loving parents, my brother, and my girlfriend, for their constant and unconditional love, care and sincere belief in me. Without their encouragement and endless support this thesis would not have been possible.

Christos Kyrkou

Dedication

This dissertation is dedicated to my father, my mother, my brother, and my girlfriend for their constant support and unconditional love. This wouldn't be possible without you.

Christos Kyrkou

Publications

Journal publications stemming from this thesis

- [J1] **Christos Kyrkou** and Theocharis Theocharides, "SCoPE: Towards a Systolic Array for SVM Object Detection", *IEEE Embedded System Letters*, vol. 1, no. 2, pp. 46-49, August 2009.
- [J2] **Christos Kyrkou** and Theocharis Theocharides, "A Flexible Parallel Hardware Architecture for AdaBoost-Based Real-Time Object Detection", *IEEE Transactions on Very Large Scale Integration (TVLSI) Systems*, vol.19, no.6, pp.1034-1047, June 2011.
- [J3] **Christos Kyrkou** and Theocharis Theocharides, "A Parallel Hardware Architecture for Real-Time Object Detection with Support Vector Machines", *IEEE Transactions on Computers*, vol.61, no.6, pp.831-842, June 2012.
- [J4] **Christos Kyrkou**, Christos Ttofis, and Theocharis Theocharides, "A Hardware Architecture for Real-Time Object Detection Using Depth and Edge Information", *ACM Transactions on Embedded Computing Systems*, vol. 13 no. 3 pp. 54:1-54:19, December 2013.
- [J5] **Christos Kyrkou**, Christos-Savvas Bouganis, Theocharis Theocharides, Marios Polycarpou " Embedded Hardware-Efficient Real-Time Classification with Cascade Support Vector Machines ", (Under Submission).

Conference proceedings stemming from this thesis

- [C1] **Christos Kyrkou**, Christos Ttofis, Theocharis Theocharides, "Depth-Directed Hardware Object Detection", *Design Automation & Test in Europe Conference & Exhibition (DATE)*, 2011, 14-18 March 2011.
- [C2] **Christos Kyrkou**, Christos Ttofis, Theocharis Theocharides, "FPGA-Accelerated Object Detection using edge information", *International Conference on Field Programmable Logic and Applications (FPL)*, pp.167-170, 5-7 Sept. 2011.

[C3] Alina Gavrijaseva, Ago Mölder, Olev Märtens, **Christos Kyrkou**, Theocharis Theocharides "Cross-Correlation-based Image Matching of Coins", The Baltic Electronics Conference (BEC), Tallinn, Estonia, October 3-5, 2012. IEEE, 319 - 322, 2012.

[C4] **Christos Kyrkou**, Christos-Savvas Bouganis, Theocharis Theocharides, "FPGA-based Acceleration of Cascaded Support Vector Machines for Embedded Applications", 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2013), 11-13 February 2013.

[C5] **Christos Kyrkou**, Christos-Savvas Bouganis, Theocharis Theocharides, "A Hardware-Efficient Architecture for Embedded Real-Time Cascaded Support Vector Machines Classification", ACM/IEEE Great Lakes Symposium on VLSI - GLSVLSI 2013, May 2-4 2013.

[C6] **Christos Kyrkou**, Christos-Savvas Bouganis, Theocharis Theocharides, "An Embedded Hardware-Efficient Architecture for Real-Time Cascade Support Vector Machine Classification", International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), Samos, Greece, pp. 129-136, 15-18 July 2013.

Other publications

[O1] **Christos Kyrkou** and Theocharis Theocharides, "Neural Network-Based Face Detector Implementation on a Virtex2 Pro FPGA Platform", Proceedings of the 3rd Greek National Student Conference of Electrical and Computer Engineering, page 62, Salonica, Greece, April 2009.

[O2] Christos Ttofis, **Christos Kyrkou**, Theocharis Theocharides and Maria. K. Michael, "FPGA-Based NoC-Driven Sequence of Lab Assignments for Manycore Systems", in the Proceedings of the IEEE International Conference on Microelectronic Systems Education (MSE 2009), co-located with the 46th Design Automation Conference, San Francisco, USA, July, 2009. *Best Paper Award*.

CONTENTS

CHAPTER 1 INTRODUCTION	1
1.1 INTELLIGENT EMBEDDED VISION: SYSTEMS THAT SEE AND UNDERSTAND	1
1.2 VISUAL OBJECT DETECTION SYSTEMS	2
1.3 THESIS SCOPE AND CONTRIBUTIONS: ENABLING EMBEDDED VISUAL OBJECT DETECTION SYSTEMS.....	3
1.4 ORGANIZATION OF THE THESIS.....	6
CHAPTER 2 GENERAL BACKGROUND, THEORY AND ALGORITHMS	9
2.1 SUPERVISED LEARNING MACHINES	9
2.1.1 <i>Monolithic Classifiers</i>	11
2.1.2 <i>Cascade Classification Scheme</i>	13
2.2 INTELLIGENT EMBEDDED VISION - VISUAL OBJECT DETECTION	15
2.3 CHALLENGES AND TRADE-OFFS FOR EMBEDDED VISUAL OBJECT DETECTION	18
2.3.1 <i>Sliding Window Search Process</i>	18
2.3.2 <i>Pattern Recognition Classification Algorithm</i>	20
2.3.3 <i>Overview of Computing Platforms for Embedded Object Detection</i>	20
2.3.4 <i>Applications of Visual Object Detection Systems</i>	28
2.4 VIOLA AND JONES FRAMEWORK FOR RAPID OBJECT DETECTION.....	29
2.4.1 <i>AdaBoost Boosting Algorithm</i>	29
2.4.2 <i>Haar-Features and Integral Image Representation</i>	31
2.4.3 <i>Haar-Feature AdaBoost-Based Cascade Classifier</i>	33
2.4.4 <i>Parallelism Opportunities and Computational Trade-offs</i>	36
2.5 SUPPORT VECTOR MACHINES OVERVIEW	36
2.5.1 <i>Support Vector Machine Formulation - Training</i>	37
2.5.2 <i>Support Vector Machine Classification</i>	44
2.5.3 <i>Computational Challenges of Support Vector Machines</i>	45
2.6 COMPLIMENTARY MATERIAL.....	50
2.6.1 <i>Feature Extraction</i>	50
2.6.2 <i>3D Stereo Computer Vision</i>	56
CHAPTER 3 A FLEXIBLE PARALLEL HARDWARE OBJECT DETECTION ACCELERATOR FOR THE ADABOOST- BASED HAAR-FEATURE CASCADE.....	59
3.1 RELATED WORK ON HAAR-FEATURE ADABOOST-BASED CLASSIFIER CASCADE IMPLEMENTATIONS	59
3.2 MAPPING ALGORITHM TO HARDWARE.....	63

3.2.1	<i>Opportunities for parallelism</i>	63
3.2.2	<i>Hardware Architecture Requirements and Mapping</i>	64
3.3	PROPOSED HARDWARE ARCHITECTURE	66
3.3.1	<i>Image Pyramid Generation (IPG) Unit</i>	67
3.3.2	<i>Integral Image and Haar-Feature Processing Array</i>	69
3.4	EXPERIMENTAL METHODOLOGY AND EVALUATION RESULTS	78
3.4.1	<i>Performance, metrics, limitations and constraints</i>	79
3.4.2	<i>FPGA Implementation and Emulation</i>	81
3.4.3	<i>ASIC Implementation and Evaluation</i>	85
3.4.4	<i>Discussion</i>	89
3.5	CONCLUSION	90
CHAPTER 4 HARDWARE ACCELERATION OF SUPPORT VECTOR MACHINES		93
4.1	RELATED WORK ON ACCELERATION OF SVMs	93
4.2	HARDWARE ACCELERATION OF MONOLITHIC SVMs	98
4.2.1	<i>Array Processing Hardware Architecture</i>	99
4.2.2	<i>Flow of Operation</i>	105
4.2.3	<i>Scalability and Implementation Issues</i>	107
4.2.4	<i>Multiclass Classification Support</i>	107
4.2.5	<i>Experimental Methodology and Evaluation Results</i>	109
4.2.6	<i>Discussion and Impact</i>	119
4.3	HARDWARE ACCELERATION OF CASCADE SVMs	119
4.3.1	<i>Challenges in the Acceleration of Cascade Support Vector Machines</i>	120
4.3.2	<i>Hybrid Hardware Architecture and Optimization Approaches</i>	122
4.3.3	<i>Experimental Platform and Results</i>	136
4.4	CONCLUSIONS	147
CHAPTER 5 REAL-TIME HARDWARE ACCELERATION OF OBJECT DETECTION USING DEPTH AND EDGE INFORMATION		149
5.1	DEPTH- AND EDGE-DIRECTED SEARCH SPACE REDUCTION	150
5.1.1	<i>Depth Extraction and Object-Size Estimation</i>	151
5.1.2	<i>Edge-Based Window Rejection Process</i>	152
5.1.3	<i>Depth- and Edge- Accelerated Object-Detection Process</i>	153
5.2	HARDWARE VISUAL OBJECT DETECTION SYSTEMS	155
5.3	HARDWARE ARCHITECTURE FOR DEPTH- AND EDGE- BASED OBJECT DETECTION ACCELERATION	159

5.3.1	<i>Edge Computation Core</i>	160
5.3.2	<i>Disparity Computation Unit</i>	162
5.3.3	<i>Window Extraction Unit</i>	164
5.3.4	<i>Classification Engine</i>	166
5.4	EXPERIMENTAL METHODOLOGY AND RESULTS	167
5.4.1	<i>Experimental FPGA Platform and Methodology</i>	167
5.4.2	<i>FPGA Implementation Discussion</i>	169
5.4.3	<i>Performance Results and Discussion</i>	171
5.5	CONCLUSIONS	176
CHAPTER 6 CONCLUSION		177
6.1	OVERVIEW AND CONCLUDING REMARKS	177
6.2	FUTURE RESEARCH DIRECTIVES	178
6.2.1	<i>Short Term Research - Pre Processing/Post Processing & Potential Improvements</i>	178
6.2.2	<i>Long Term Research</i>	183
APPENDIX A: A CASE STUDY FOR EDGE-ACCELERATED OBJECT DETECTION		187
APPENDIX B: IMPACT OF FEATURES ON SVM CLASSIFICATION		193
APPENDIX C: A NOTE ON THE HARDWARE IMPLEMENTATION OF ARTIFICIAL NEURAL NETWORKS		197
REFERENCES		203

Christos Kyrkou

LIST OF FIGURES

FIGURE 2-1. SUPERVISED LEARNING WITH MONOLITHIC CLASSIFIERS	12
FIGURE 2-2. SUPERVISED LEARNING WITH CASCADE CLASSIFIERS	14
FIGURE 2-3. OBJECT DETECTION PROCESS	18
FIGURE 2-4. ARCHITECTURE OF A FIELD PROGRAMMABLE GATE ARRAY (FPGA)	24
FIGURE 2-5. A HETEROGENEOUS EMBEDDED VISION SOC.....	25
FIGURE 2-6. OUTLINE OF THE ADAPTIVE BOOSTING (ADABOOST) ALGORITHM.....	30
FIGURE 2-7. EXAMPLES OF HAAR-LIKE FEATURES	32
FIGURE 2-8. INTEGRAL IMAGE AND FAST FEATURE COMPUTATION.....	33
FIGURE 2-9. VIOLA AND JONES ALGORITHM FLOW	34
FIGURE 2-10. SUPPORT VECTOR MACHINE MAIN CONCEPTS	37
FIGURE 2-11. KARUCH-KUHN-TUCKER (KKT) CONDITIONS.....	39
FIGURE 2-12. KERNEL TRICK AND KERNEL CONSTRUCTION	41
FIGURE 2-13. THE SMO ALGORITHM	44
FIGURE 2-14. SUPPORT VECTOR MACHINE CLASSIFICATION PROCEDURE.....	46
FIGURE 2-15. REDUCED-SET METHOD	48
FIGURE 2-16. LOCAL BINARY PATTERN CODE GENERATION PROCESS	51
FIGURE 2-17. LOCAL BINARY PATTERN HISTOGRAM GENERATION PROCESS	52
FIGURE 2-18. EDGE DETECTION PROCESS	53
FIGURE 2-19. HISTOGRAM OF ORIENTED GRADIENT	54
FIGURE 2-20. HISTOGRAM EQUALIZATION.....	55
FIGURE 2-21. STEREO SETUP AND DEPTH COMPUTATION.....	57
FIGURE 3-1. PARALLELIZATION IN THE ADABOOST-BASED OBJECT DETECTION	63
FIGURE 3-2. SYSTOLIC-ARRAY INSPIRED ACCELERATOR ARCHITECTURE WITH THE TWO MAJOR UNITS	67
FIGURE 3-3. IMAGE PYRAMID GENERATION UNIT	68

FIGURE 3-4. SYSTOLIC PROCESSING ARRAY ARCHITECTURE AND COMPONENTS.....	70
FIGURE 3-5. COLLECTION AND COMPUTATION UNIT.....	72
FIGURE 3-6. PROCESSING ARRAY BITWIDTH REQUIREMENTS	73
FIGURE 3-7. EVALUATION UNIT	74
FIGURE 3-8. SYSTOLIC ARRAY COMPUTATION FLOWS	76
FIGURE 3-9. FPGA SYSTEM PROTOTYPE	84
FIGURE 4-1. SUPPORT VECTOR MACHINE ARRAY PROCESSING ARCHITECTURE.....	99
FIGURE 4-2. VECTOR UNIT	101
FIGURE 4-3. SCALAR UNIT	102
FIGURE 4-4. WINDOW BUFFER REGISTER ARRAY ARCHITECTURE	104
FIGURE 4-5. TRANSFERRED DATA WORD IN THE HORIZONTAL DIRECTION	106
FIGURE 4-6. SUPPORT VECTOR MACHINE PROCESSING ARRAY MULTICLASS SUPPORT	108
FIGURE 4-7. IMPLEMENTED FPGA SUPPORT VECTOR MACHINE ARRAY PROCESSING SYSTEM.....	113
FIGURE 4-8. FPGA SYSTEM PROTOTYPE	114
FIGURE 4-9. SVM CASCADE CLASSIFICATION PROCESS OVERVIEW.....	122
FIGURE 4-10. HARDWARE REDUCTION METHOD	124
FIGURE 4-11. CASCADE RESPONSE EVALUATION METHOD	126
FIGURE 4-12. SUPPORT VECTOR MACHINE CASCADE SYSTEM ARCHITECTURE	127
FIGURE 4-13. PARALLEL PROCESSING MODULE (PPM) ARCHITECTURE	128
FIGURE 4-14. SEQUENTIAL PROCESSING MODULE (SPM) ARCHITECTURE.....	130
FIGURE 4-15. RESPONSE PROCESSING UNIT (RPU).....	132
FIGURE 4-16. LOCAL BINARY PATTERN PROCESSING HARDWARE	134
FIGURE 4-17. OPTIMIZED I/O MECHANISM	135
FIGURE 4-18. BLOCK DIAGRAM OF THE FPGA SYSTEM.....	136
FIGURE 4-19. CASCADE SUPPORT VECTOR MACHINE STRUCTURE	138
FIGURE 4-20. SUPPORT VECTOR MACHINE CASCADE EARLY STAGE RESPONSES	138

FIGURE 4-21. ADAPTATION USING THE ROC CURVE AND NEW DETECTION RATES	139
FIGURE 4-22. DETECTION RESULTS ON 800×600 IMAGES	142
FIGURE 4-23. COMPARATIVE RESULTS OF DIFFERENT CASCADE CONFIGURATIONS	143
FIGURE 5-1. NUMBER OF WINDOWS AS THE IMAGE SIZE AND NUMBER OF SEARCH SCALES INCREASE.....	150
FIGURE 5-2. WINDOW SIZE ESTIMATION USING DEPTH INFORMATION.....	151
FIGURE 5-3. OBJECT VS BACKGROUND EDGE INFORMATION	153
FIGURE 5-4. OBJECT DETECTION PROCESS USING EDGE AND DEPTH INFORMATION	154
FIGURE 5-5. DEPTH AND EDGE BASED SYSTEM ARCHITECTURE.....	160
FIGURE 5-6. SOBEL EDGE OPERATOR.....	161
FIGURE 5-7. EDGE COMPUTATION CORE.....	162
FIGURE 5-8. DISPARITY COMPUTATION UNIT.....	163
FIGURE 5-9. WINDOW EXTRACTION UNIT	164
FIGURE 5-10. DYNAMIC IMAGE DOWNSCALING THROUGH REVERSE MAPPING.....	165
FIGURE 5-11. SUPPORT VECTOR MACHINE CLASSIFIER ARCHITECTURE	166
FIGURE 5-12. STEREO CAMERA SYSTEM	168
FIGURE 5-13. REDUCTION IN THE NUMBER OF WINDOWS USING DEPTH AND EDGE INFORMATION	172
FIGURE 5-14. DEPTH AND EDGE DIRECTED FACE DETECTION RESULTS	174
FIGURE 5-15. DEPTH AND EDGE DIRECTED CAR DETECTION RESULTS.....	175
FIGURE 6-1. INTEGRATING PREPROCESSING TO SVM CLASSIFICATION ARCHITECTURE.....	179
FIGURE 6-2. ONLINE TRAINING OF CASCADE CLASSIFIERS	181
FIGURE A-1. EDGE-BASED WINDOW EXTRACTION METHOD	187
FIGURE A-2. EDGE ACCELERATION SYSTEM ARCHITECTURE	188
FIGURE A-3. DETECTION RESULTS USING EDGE AND COMPARED TO SLIDING WINDOW	191
FIGURE C-1. NEURAL NETWORK AND NEURON MODEL.....	198

Christos Kyrkou

List of Tables

TABLE 2-1 SUPPORT VECTOR MACHINE KERNEL FUNCTIONS	42
TABLE 3-1: ALGORITHM AND METHOD COMPARISONS FOR RELATED FPGA WORKS	62
TABLE 3-2. SYNTHESIS RESULTS FOR THE VIRTEX II PRO FPGA IMPLEMENTATION.....	83
TABLE 3-3. RESULTS COMPARISON OF RELATED WORK IMPLEMENTATIONS ON FPGAS	85
TABLE 3-4: DETECTION APPLICATIONS TRAINING DATA	86
TABLE 3-5: ASIC IMPLEMENTATION - SIMULATION RESULTS	88
TABLE 3-6: ASIC IMPLEMENTATION - RELATED WORK COMPARISON	89
TABLE 4-1 COMMON SVM KERNEL FUNCTIONS	100
TABLE 4-2: SUPPORT VECTOR MACHINE PROCESSING ARRAY PARAMETERS AND RESULTS	112
TABLE 4-3: ARRAY PROCESSING ENGINE FPGA SYNTHESIS RESULTS.....	113
TABLE 4-4: SUPPORT VECTOR MACHINE ARRAY HARDWARE PARAMETERS	114
TABLE 4-5: SUPPORT VECTOR MACHINE CASCADE SYSTEMS OVERVIEW	121
TABLE 4-6: CASCADE DETECTION SYSTEM PARAMETERS	139
TABLE 4-7: FPGA RESOURCE REQUIREMENTS PER UNIT AND SYSTEM	141
TABLE 4-8: STATISTICS FOR EACH CASCADE STAGE	143
TABLE 4-9: COMPARISON WITH RELATED WORK	144
TABLE 5-1: SUMMARY OF FPGA IMPLEMENTATIONS OF OBJECT DETECTION SYSTEMS	156
TABLE 5-2: SYSTEM PARAMETERS	169
TABLE 5-3: FPGA RELATED WORK SYNTHESIS RESULTS.....	171
TABLE A-1: CLASSIFICATION SYSTEMS FPGA SYNTHESIS RESULTS.....	191
TABLE C-1 COMMON ANN ACTIVATION FUNCTIONS	199

Christos Kyrkou

CHAPTER 1

INTRODUCTION

In recent years there has been a growing interest in integrating senses (e.g. hearing, touch, vision, etc.) into embedded systems in order for them to interact more intelligently with their host environment, make decisions and take actions based on the additional sensing information. An integral part of this effort is to enhance embedded systems with vision, the principle sense not only in human beings but also in almost all living animals, which has the potential to transform the way embedded systems are used in existing as well as emerging applications such as security and surveillance, automotive, medical, gaming, aerospace, retail, industrial and robotics amongst others.

1.1 Intelligent Embedded Vision: Systems that see and understand

Intelligent embedded vision refers to embedded vision systems that use intelligent algorithms, tools and technologies from the fields of computer vision, machine learning, and digital signal processing that enable computing systems to “see and understand” their environment through visual means [1]. Through the advances in the technologies of integrated chip (IC) design and manufacturing, as well as the development of advanced algorithms from the aforementioned scientific fields, it is possible to integrate embedded vision into a wide range of computing systems, including embedded systems, mobile devices, personal computers and recently the cloud. Vision-based technologies have been established in a number of markets, the most successful of which include factory automation for pharmaceuticals and automotive assembly for tasks such as quality control and packaging. These applications offer a glimpse at the potential of embedded vision applications and as the processing technologies and platforms improve there is a growing interest in the integration of vision technologies and features in embedded systems [1]. The Embedded Vision Alliance (EVA) [2], which is a consortium of companies and institutions that aims to enable rapid growth of embedded vision applications, foresees that the next few years there will be an ever increasing interest in the adoption of vision technologies. It is predicted that 600 million smartphones with vision-based gesture recognition will be shipped in 2017 (ABI Research)

[2], while there will be an annual revenue growth of 6 – 9% worldwide for special-purpose vision processors targeting automotive applications [2], reaching \$187 million by 2016 [2] (IMS Research), while augmented reality applications and technologies are expected to generate close to \$300 million in 2013 [2] (Juniper Research). These figures suggest that embedded vision technologies will become more and more common in computing systems. Thus it is anticipated that there will be a wealth of opportunities for new applications in security and surveillance, automotive and transportation systems, aerospace, defense, healthcare, as well as augmented reality applications. In many cases, the addition of vision capabilities can transform existing products. For example, incorporating gesture recognition, face detection, facial recognition and eye tracking into embedded devices will make those systems more responsive and aware of their environment and increase their interaction capabilities. In addition there will be a growing number of new embedded applications that will extend beyond the current trends, such as autonomous vehicles and human-like robots. It is anticipated that such systems will be necessary for various embedded applications that will require higher-level vision processing to perform various functions such as handwritten digits recognition for reading, vehicle detection for navigation, as well as face and human detection for "human-like" interaction as well as security and surveillance purposes.

1.2 Visual Object Detection Systems

Visual object detection is a fundamental task towards higher-level vision and refers to the ability of a computing system to analyze an image/video in order to determine the presence of an object(s) of interest. This is a necessary step towards image understanding and decision making. Humans have the ability to recognize and identify a large number of objects very fast and with very high accuracy. As such, research in visual object detection over the past twenty years has strived to develop algorithms and methods to allow computers to perform object detection as efficiently as humans, and for some well studied applications such as face detection [3],[4] and pedestrian detection [5] these algorithms have demonstrated very good accuracies. However, they are still lacking in terms of processing efficiency and performance compared to the human visual system. This aspect is crucial in order to be able to integrate object detection tasks into embedded applications which operate under real-time and low-

power constraints. As such, software implementations of object detection for embedded applications, even if highly flexible, have difficulties in satisfying these constraints due to the lack of available processing resources [6], [7], [8]. Consequently, hardware acceleration has gained considerable interest as a way to meet the imposed real-time and power constraints [1], [9],[10] and perform efficient visual object detection for embedded applications. With hardware acceleration it is possible to use a specialized parallel digital hardware architecture to perform these functions faster and more efficient than it is possible in software running on the general-purpose processing platforms [11].

Common hardware acceleration platforms that have been used for visual object detection applications include specialized Digital Signal Processors (DSPs), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) [1]. General purpose computing capabilities for embedded GPUs are not yet widely supported on embedded platforms [12], as a result current generations of General Purpose GPUs (GPGPUs) that are found as components on PCs are difficult to be used in embedded environments due to power consumption issues [11]. On the other side Digital Signal Processors (DSPs) are optimized for low-power operation and for executing a small code footprint for a large amount of data [1]. However, the fixed logic and low number of processing cores prohibit this platform from offering the required parallelism for real-time performance. FPGAs allow the design of an application specific architecture with true parallel execution which can also be later be implemented as an Application Specific Integrated Circuit (ASIC). Hence, FPGAs have emerged as an attractive and capable platform to trade-off the various constraints found in embedded applications and such are becoming a popular solution for the hardware acceleration of embedded vision systems [1], [13].

1.3 Thesis Scope and Contributions: Enabling Embedded Visual Object Detection Systems

Hardware acceleration through custom parallel hardware architectures is a promising means towards efficient embedded visual object detection systems in order to address the various constraints of embedded applications. However, at the same time it poses many challenges as hardware architectures must keep a low area overhead by using the minimum

amount of hardware resources for the specific functionality to keep both size and power at low levels, while also providing the required real-time performance (30 FPS for video, need to be higher for applications requiring higher response rates), maintain the desired accuracy, and operate within the allowed power budget. Furthermore, the same hardware design may not be suitable for all embedded applications as each has its own set of specifications and platform implementation. As such, hardware architectures for embedded object detection must have a modular and a regular design to maintain scalability, and also should provide a degree of run time reconfigurability to handle different embedded applications. Finally embedded hardware architectures for object detection must provide efficient memory management, with parallel memory access and predictable access patterns in order to facilitate high performance.

Hence, the research in this thesis is concerned with the design and development of efficient generic hardware accelerators for visual object detection which can be used in embedded environments. The most popular and successful approaches for object detection are appearance-based methods which use example images (called templates or exemplars) of the objects to train a supervised learning algorithm to recognize the desired object [14]. Appearance-based methods have shown significantly superior performance over other methods. Thus this thesis investigates the hardware acceleration of two widely used approaches [3] for object detection based on machine learning classification algorithms namely the Viola and Jones AdaBoost haar-feature cascade classifier [15], and Support Vector Machines (SVMs) [16]. Both are widely used and popular approaches and have attracted interest for different reasons, the former because of its computational efficiency [3] and the latter for its solid mathematical foundations and accuracy results as well as its generic approach that makes it a very useful tool for vision applications [17].

The Viola-Jones rapid object detection framework is considered a seminal work towards real-time 2D object detection for a wide range of applications and platforms [3]. The framework has been implemented in a wide range of software libraries including Intel's OpenCV [18] as well as on FPGAs using hardware architectures. However, existing architectures rely on parallelizing the processing of only a single window and thus require more hardware to achieve higher performance as the image size increases or if they need to process more windows in parallel. Also, such architectures are fixed for a specific application

training set and thus are not flexible enough to be used in other applications. The architecture for the Viola-Jones detection framework relies on a scalable systolic array processing architecture which processes many windows in parallel which yields extremely high detection frames per second (FPS) compared to reported works and is also able to process different training sets thus also providing flexibility. As the array elements are modular and simple, and communication is regular and predetermined, the architecture is highly scalable and can operate on high frequency. The designer can select all the appropriate design parameters with the targeted operating environment in mind, without affecting the real-time constraints. The architecture was prototyped on an FPGA platform, and also synthesized for ASIC design flow for high-end applications. Through both the FPGA emulation and large-scale implementation the architecture can detect objects in large images (up to 1024×768 pixels) with frame rates that can vary between 64 – 139 fps for various applications and input image frame sizes. In addition, the hardware implementation achieves similar detection rates to the equivalent software implementation.

SVMs have demonstrated excellent classification accuracies in a wide range of tasks, including generic object detection [19],[20]. However, because of the computational intensive operations involved they have not been considered for real-time embedded object detection. As such, SVM-based hardware object detection systems proposed application specific architectures for small scale problems (low number of Support Vectors (1-100) or and low dimensionality (8-256)). Overall, there are limited proposed hardware architectures that offer the flexibility and parallelism capabilities to be used in a variety of object detection applications. In order to tackle larger scale problems this thesis presents an array-based processing architecture which provides parallel processing, resource sharing amongst the processing units, and efficient memory management. Furthermore, the size of the array-based architecture is scalable to the hardware demands, and can also handle a variety of applications such as multi-class classification problems. The proposed architecture is integrated into an object detection system that is implemented on an FPGA and is evaluated using three object detection applications: face, pedestrian and car side view detection. Results indicate high performance in terms of frame rate (40 – 122 FPS for a variety of applications) and detection accuracy (76 – 78%) for the three benchmark applications. Also, the presented work in this thesis is one of the first to address a hardware architecture for cascade SVM classification.

Existing hardware architectures consider only single SVM classifiers which are not optimized to efficiently handle problems where the majority of data belongs to one of the two classes, such as object detection. Cascade classification schemes can offer speedups over single SVMs. As such, a hybrid architecture is presented that is able to process cascade SVMs and enables real-time classification ~40 FPS for 800×600 resolution images.

In addition to the hardware acceleration of intelligent machine learning algorithms for object detection, this thesis also presents how additional mechanisms can be integrated into such systems in order to improve performance in an intelligent manner. Typically, such approaches relied on motion detection and/or object color information, however, these do not address the limiting factor in the performance of object detection systems which is to find the sizes of the objects in the image. As such this thesis also presents an alternative to the traditional sliding window search approach based on 3D depth and edge information that can be integrated into object detection accelerators to recognize the object size and background regions respectively, thus improving performance and increasing the awareness of object detection systems. A dedicated hardware is proposed that integrates together edge image processing, 3D processing and classification in order to provide an efficient and generic framework for real-time object detection. The proposed architecture is implemented on an FPGA platform targeting a face detection application, and is able to handle various image sizes of 320×240 , 640×480 , and 800×600 pixels, and the introduction of these methods results in an average reduction by $\sim 7.4 \times$ in the number of data that needs to be processed by the SVM classification system, achieving 271, 42, and 23 frames-per second (FPS) respectively, while also providing a 52% reduction in the false positive rate, compared to traditional object detection search methods.

1.4 Organization of the Thesis

This thesis is organized as follows. First, relevant topics to the research conducted in this thesis as well as background material and related algorithms are discussed in Chapter 2. These include a general discussion on classification and pattern recognition approaches and a more detailed look at the theory of support vector machines and the Viola and Jones object detection framework, as well as other complementary material such as computing platforms and

preprocessing methods. The related work on implementations of object detection systems for the two classification algorithms are presented separately in each respective chapter.

A real-time hardware architecture for the Viola and Jones object detection framework is presented in Chapter 3. It is based on a systolic-array inspired architecture which offers massive parallel processing for multiple search windows which enables it to offer real-time performance of ~ 64 FPS for 320×240 images.

In Chapter 4 two hardware architectures for SVM-based object detection are presented. The first target monolithic (single) SVM processing and facilitates the parallel processing of search windows in order to boost classification performance, achieving 40 – 120 FPS for a variety of applications. The second is optimized for cascade classification where different classifiers have different computational demands and are thus implemented with different architectures. Furthermore, a hardware reduction method is proposed which reduces the amount of computational resources needed, and a meta-learning approach is used to improve performance resulting in 40 FPS for 800×600 images.

An alternative approach to the traditional sliding window search is presented in Chapter 5. It is based on using 3D depth information to find the size of a possible object thus avoiding the image downscaling process. In addition, edge information is used to discard image regions which do not contain many edges (i.e. background regions). This approach manages to offer real-time performance (23 – 271 FPS) for different image sizes.

Chapter 6 draws conclusions and summarizes the main contributions and outcomes of this thesis and discusses some ideas for the potential short-term as well as long-term directions of this research.

Appendix A describes a method to accelerate object detection using only edge information, while Appendix B shows how different features and preprocessing methods impact the accuracy of the Support Vector Machine classification method. Finally, Appendix C, provides some details on artificial neural networks.

Christos Kyrkou

CHAPTER 2

GENERAL BACKGROUND, THEORY AND ALGORITHMS

This chapter provides an overview on important background material regarding supervised machine learning algorithms and cascade classifiers in general, visual object detection in images and the computing platforms used for evaluation of approaches, as well as more specific material regarding AdaBoost-based cascade Haar-classifiers, Support Vector Machines (SVMs), and other complementary material that have been used in the research done in this thesis. In addition, it also provides details on related subjects that have not been directly used in this thesis but can be used as alternative approaches.

2.1 Supervised Learning Machines

As the applications and demands for ambient intelligence and autonomous systems become more prominent, embedded computing systems are applied to solve ever more complex problems. In many cases, the method of computing the desired output from a set of inputs might be computationally very expensive and infeasible, or even in some cases not known [14]. Hence, tasks cannot be solved using an algorithm, which is a set of instructions and steps that should be carried to find a given output for a set of inputs. As an example, one might consider the case where it is necessary to tell spam emails from genuine emails. An email is essentially a set of characters but we do not have a way to systematically and analytically go from the input to the output which is a yes/no answer to the question, is this email spam?

Machine learning concerns the development of alternative approaches that can help computers acquire knowledge (*learn*) from a given set of data, in order to be able to solve more complex problems. A specific approach called *supervised learning* [21] relies on *learning – by – examples*, that in a way is similar to the human learning process, where the computer is presented with input/output pairs and attempts to *learn* the input/output functionality from examples without being given exact specifications of how the output is generated. The examples of input/output pairs are referred to as the *training data/set* [14]. Given a labeled training set $D = \{(x_i, y_i) \mid x_i \rightarrow \text{data sample}, y_i \rightarrow \text{target value}\}$, a supervised learning algorithm tries to compute a mapping function/model f such that $f(x_i) =$

y_i for sample i in the training set. So in the case of spam emails for example the training set would consist of emails that we know beforehand are either spam or genuine. These will be examined to find patterns that constitute each category in order to determine an appropriate mapping between the input which is a set of characters and the output which is the predetermined classification label. This mapping function describes the relationship between the data samples and their respective class labels, and is used to classify new unknown data, or in the specific example new incoming emails. The process of obtaining the mapping function/model is called the *training phase*, and is usually an optimization problem with certain objectives. The algorithm used to solve it is called the *training algorithm*. The training results in a function/model with corresponding *training data/features* (weights, thresholds, etc.). This model can take different forms and structures depending on the specific machine learning approach which can consist of if-then-else or other decision rules predetermined by an expert, layers of interconnected weighted processing elements, majority voting schemes, or a linear combination of weighted data. Once this model is obtained we can use that model to predict the label/classification result for a data point which has not been observed before. This process is called *testing/prediction* and is the process that needs to be implemented by systems and devices that will be used in the targeted classification application.

A learning problem with binary outputs (a two-class classification problem) is referred to as a *binary classification* problem and the target labels y_i have only two values (either 0 and 1 or -1 and 1 depending on the problem formulation). An example, is the spam email classification problem considered thus far. A learning problem with a finite number of categories/classes is called *multi-class classification* ($y_i = 0, 1, \dots, n - 1$, where n is the number of classes) and one example is when we want to recognize each of the characters inside an email. Finally, when the target values y_i correspond to real-numbers the problem becomes known as *regression* and it can be used to predict option prices in the stock market for example. This thesis is primarily concerned with binary classification problems. Such classification problems often use either monolithic (single) classification algorithms to learn the association between data and their labels or cascade (multistage) classification schemes

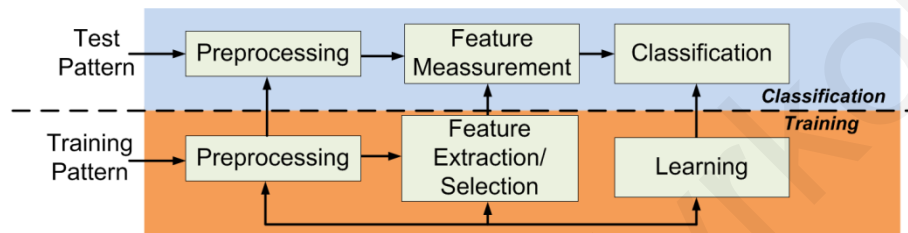
that utilize multiple single classifiers of increasing complexity. Details for these types of classifiers are provided next.

2.1.1 Monolithic Classifiers

Monolithic classifiers consist of a single supervised learning algorithm and subsequent classification model, obtained after training, which is used to classify all the input data. In both the training and classification procedures (Figure 2-1) preprocessing of the training and test sets may need to take place and also meaningful features may need to be extracted. These are issues which will be discussed in more detail in the succeeding Sections. The main characteristic of monolithic classifiers is that all inputs go through the same classification process. Training of a supervised learning algorithm first requires constructing the training set which contains examples of positive data (whose target label y_i is +1) and negative data (whose target label y_i is either 0 or -1). Second, it is necessary to select the parameters of the classification algorithm. These parameters determine how well the underlying algorithm will fit to the training set and also the form of the decision boundary where one side corresponds to positive samples and the other to negative samples. Most importantly, however, these parameters also control how well the classification algorithm will generalize, i.e. how well it will be able to classify new unseen data. These parameters and the resulting training data/features derived from the training phase constitute the classification model and also affect the amount of algorithmic-specific operations that must be carried out in order to classify new data. The number of parameters that need to be determined and the size of the training set impact the overall complexity of a classifier and consequently its performance in terms of classification accuracy and classification speed.

The accuracy of a classifier is measured using a *test set* of samples that have not been used for training. This is information captured in what is called the *confusion matrix*. The confusion matrix (illustrated in Figure 2-1) displays four values: the *true positive (TP)* rate, the *true negative (TN)* rate, the *false positive (FP)* rate, and the *false negative (FN)* rate. Ideally a classification algorithm would exhibit a 100% TP rate and a 0% FP rate. However, since the classification model is unknown and the machine learning algorithm tries to find an approximation of that model based on a finite set of samples, these perfect

classification rates are not possible. However, the objective of the training phase is to obtain detection rates which are close to those figures. Better detection rates can be achieved by increasing the complexity of the classification algorithm either by increasing the number of its free parameters, by enlarging the training set, or by extracting features from the raw data samples that would aid the training process in achieving a better discrimination, or by a combination of these approaches. Appendix B illustrates how feature extraction approaches can aid the classification algorithm in obtaining better results. Also Section 2.6.1 provides more details on feature extraction methods.



(a)

	Predicted "POSITIVE"	Predicted "NEGATIVE"	
True Class "POSITIVE"	TP: Number of samples predicted as <i>positive</i> when true class is <i>positive</i>	FN: Number of samples predicted as <i>negative</i> when true class is <i>positive</i>	<div style="display: flex; flex-direction: column; align-items: center;"> <div style="border: 1px solid black; width: 20px; height: 20px; background-color: yellow; margin-bottom: 5px;"></div> Erroneous Detections </div> <div style="display: flex; flex-direction: column; align-items: center; margin-top: 10px;"> <div style="border: 1px solid black; width: 20px; height: 20px; background-color: lightgreen; margin-bottom: 5px;"></div> Correct Detections </div>
True Class "Negative"	FP: Number of samples predicted as <i>Positive</i> when true class is <i>Negative</i>	TN: Number of samples predicted as <i>Negative</i> when true class is <i>Negative</i>	

(b)

Figure 2-1. Supervised learning with monolithic classifiers

(a) Training and Classification for a monolithic classifier. (b) Confusion Matrix

The performance of a classifier depends on the interrelationship between the training set size, the number and type of features as well as classifier complexity. It has been often observed that the added features may actually degrade the performance of a classifier if the number of training samples that are used to design the classifier is small relative to the number of features [21]. However, increasing the training set size means that the classifier complexity will also increase either because more samples will be selected from the training set or because the number of free parameters of the classifier will need to increase to accommodate the larger training set. In both cases the trade-off for higher accuracy is reduced classification

speed. The classification speed of high accuracy monolithic classifiers can be improved by utilizing a hierarchy of classifiers.

2.1.2 Cascade Classification Scheme

The detection accuracy of a monolithic classifier can be improved by increasing the training set size and the capacity of the model to allow it to better fit the data thus achieving higher true positive rates and lower false positive rates. However, the major drawback of such classifiers is that they will require more time to compute the classification outcome for new data. Furthermore, the higher complexity is required to discriminate between samples that belong to different classes but exhibit a high degree of similarity. In contrast however, there are many samples that can be distinguished by classifiers of lower complexity since the samples have distinctively different features. The cascade classification scheme tries to take advantage of this very observation by building a hierarchy of classifiers with different training data and feature requirements in order to provide increased detection performance while radically reducing computation time. Thus the early classifiers in the hierarchy focus on discriminating between the easily distinguished samples while the latter focus on the more difficult samples that exhibit a high degree of similarity. As such, cascade classification schemes attempt to reject as many negative data as possible at the earliest stage possible. While only a positive instance will trigger the evaluation of every classifier $\mathcal{C}(i)$ in the cascade, which happens only for a small fraction of the input data.

Training of cascade classifiers involves a tradeoff between the number of cascade stages, the number of features in each stage and the threshold of each stage. Due to the difficulty in finding the optimum solution to this problem a simple yet efficient approach is usually followed to construct a cascade of classifiers and is outlined in Figure 2-2. A stage in the cascade is first trained using the desired supervised machine learning algorithm until the desired detection rates for that stage have been reached. Then the whole cascade classifier is evaluated using a test set that determines the stopping condition. Another stage is added to the hierarchy, with increased complexity (i.e. more features and more training data), if the overall detection rates of the cascade are not sufficient. This process is repeated until the maximum number of stages have been reached or if the overall targeted accuracy has been achieved. The

negative training set used to train each additional stage is filtered each iteration by first passing negative samples to the current cascade and collecting the false detections (i.e. those that have been classified as positive). This results in reduced training time for each additional stage and allows the additional stage to focus on the "difficult" examples that cannot be distinguished by simpler models.

-
- Given a training set of P_{Tr} positive samples, and N_{Tr} negative samples and a test set of P_{Te} positive samples, and N_{Te} negative samples.
 - Do
 - i. Select a random subset of the negative training set $N_{Tr} \rightarrow N_{Tr}^{rand}$.
 - ii. Train a classifier $C(i)$ using P_{Tr} and N_{Tr}^{rand} .
 - iii. Get accuracy of classifier $C(i)$ using P_{Te} and N_{Te} .
 - iv. Adjust threshold of classifier $C(i)$ to minimize false negatives (FN).
 - v. Screen the negative training set N_{Tr} and remove the true negatives (TN) from the training set resulting in a new negative training set N'_{Tr} .
 - vi. Substitute the training set $N'_{Tr} \rightarrow N_{Tr}$.
 - vii. Evaluate classifier cascade using P_{Te} and N_{Te} and stop if accuracy is satisfactory. Else add an n^{th} stage and repeat process.

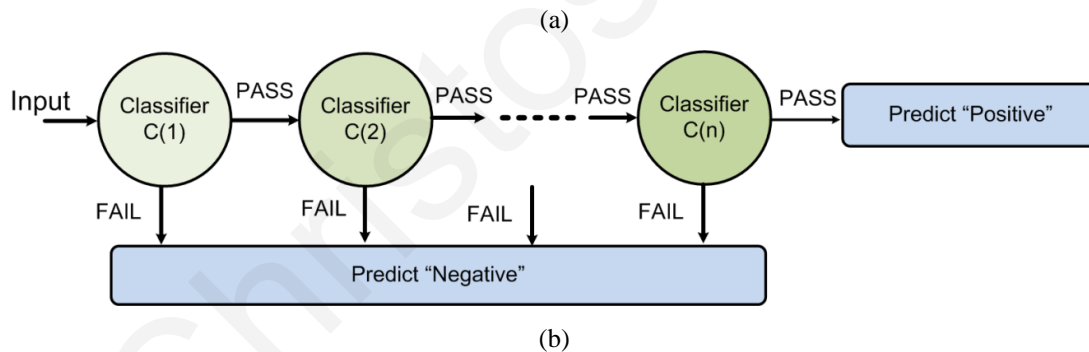


Figure 2-2. Supervised learning with cascade classifiers

(a) Cascade classifier construction overview (b) Cascade classification scheme

The cascade approach has been one of the mostly widely used schemes for problems where one class appears more frequently than the other. It manages to speedup both the training as well as the classification process which are both equally important the latter to allow for trying out new methods and the former to meet real-time detection constraints. Hence, cascade

classification schemes are widely used for real-time embedded applications, such as visual object detection, where performance speed is a critical constraint. The cascade classification scheme is only one of the ways to combine different classifiers together. Depending on the application different ensembles/structures can be utilized to combine the classifiers. For example, tree classifiers can be used for multi-class applications where each path triggers a different class. Majority voting schemes can be used to combine different classification algorithms and obtained a more accurate and robust classification result rather than improved detection speeds.

2.2 Intelligent Embedded Vision - Visual Object Detection

The detection/discovery of visual objects is a perceptual and cognitive task fundamental to vision and intelligence. It can be useful for a wide range of embedded applications ranging from robotics, surveillance and census systems, human-computer-interaction, intelligent transport systems, and military. Hence, the realization of intelligent embedded vision systems that can perform visual object detection is critical for such applications.

The process of visual object detection deals with determining whether an object of interest is present in an image/video frame or not: regardless of its size, orientation, and the environmental conditions which is found in. The high degree of variability makes it difficult to describe an object analytically by following an algorithmic step by step approach. Hence, object detection is typically viewed as a machine learning pattern recognition problem where the goal is to given an image to classify it as an object or non-object. There are different methods used to perform object detection the most notable of which are [22],[23],[17]:

Knowledge-based methods: These techniques are based on rules that codify the human knowledge about the object of interest and its characteristics.

Feature invariant techniques: These methods consists of finding structural object features that remain invariant regardless of pose, lighting conditions, or viewpoint.

Template matching methods: Several standard patterns/models are stored to describe the object as a whole or as different components. The correlations between the stored models and input are computed to perform detection.

Appearance-based methods: In contrast to template-based methods the models are learned by examples of objects and non-objects, through supervised machine learning algorithms (Support Vector Machines, Neural Networks, etc.), which find relationships between data instances and classes to capture the variability of visual appearance.

Obviously, the above methods are interrelated and can be used together in order to provide higher and more robust detection accuracies. Appearance-based methods constitute the most popular detection approaches [22] but they are often combined with knowledge-based or feature invariant techniques to improve detection performance. These types of methods obtain good results due to the fact that they can generalize well given that the variability in the object appearance can be captured by the given training set and the chosen features offer adequate descriptive capabilities. Moreover, they incur a lower computation cost compared to other methods.

The overall visual object detection process begins by first receiving an input image/video frame from a camera or other adequate image source, which subsequently will then be searched in order to find possible objects of interest. This search is done by extracting smaller regions from the frame, called *search windows*, of $m \times n$ pixels, which are processed by a classification algorithm to determine if they belong to the object of interest class or not [9]. The search window size is such so that it corresponds to the size of the object of interest. Thus, the classification algorithm learns to categorize search windows of a particular size. However, the object of interest may appear in the image/video frame at a larger size than the size of the search window. In such a case, the classification algorithm will not be able to detect the object. To account for this scenario an object detection system may either increase the size of the search window, or decrease the size of the input image (downscaling), effectively reducing the size of the object of interest, and then reexamines the downscaled image with the same search window size. The latter process is often preferred as it is more efficient [58] as the former requires training many classifiers, one for each window size, and also to process large images as the window size increases. On the other hand, the former approach requires training only a single classifier for the targeted window size. The downscaling process is done in steps to account for various object sizes, down to the size of the search window and scaling happens by mapping old coordinates to new ones using a *scaling factor* as shown in Equation 2-1.

Hence, many downscaled images are produced from a single input image/video frame, each in turn producing a number of search windows, which increases the amount of data that must be processed by the classification algorithm. Search windows can be extracted from every pixel location in the image (exhaustively) or every few pixels. The term which determines the distance between successive search windows is called the *window pixel step*. This window step is application specific and is relative to the size of the object of interest. Small objects can appear within a distance of a few pixels between them and as such, usually a small window step is chosen, whereas for larger search windows the window step can be increased.

$$\begin{pmatrix} X_{new} \\ Y_{new} \end{pmatrix} = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \times \begin{pmatrix} X_{old} \\ Y_{old} \end{pmatrix} \quad \text{Equation 2-1}$$

Each window that is extracted from the image is processed to account for different lighting conditions and other environmental variations, or to extract meaningful features which are used for classification. These features can either be shape, color, intensity, and responses of various filters and feature extraction algorithms (edges, local binary patterns, Haar wavelets, histograms, etc.). Using features makes the detection process more robust since it provides a more representative description of the object and reduces the within-object-class variability. However, the addition of feature extraction approaches and preprocessing methods can have a negative effect on the classification speed even though the accuracy can be improved. An illustration of how features can improve the classification performance is given in Appendix B, while different features are discussed in Section 2.6.1.

It is important to consider the metrics used to measure the performance of an object detection system. An image object detection system is characterized by how accurately it can classify data as well as how many image frames it can process per second. Thus, the two commonly used performance metrics are the *detection accuracy*, and *frames per second (FPS) or frame rate*. Detection accuracy is usually measured on a given *test set* where the expected outcome for a sample is compared to the actual outcome of the object detection system. The detection accuracy is the percentage of samples for which the expected outcome matches the actual outcome of the detection system. FPS concerns the throughput of a system and is the maximum number of digital video/image frames, of a given size, that the detection system can process in one second. A minimum performance of 30 FPS is required in order for an object detection system to be capable for *real-time* video processing

[58]. However, depending on the application higher frame-rates may be necessary thus higher system performance is needed. This is typically the case if other image processing and recognition algorithms have to coexist with detection, or if multiple video feeds from different sources need to be processed.

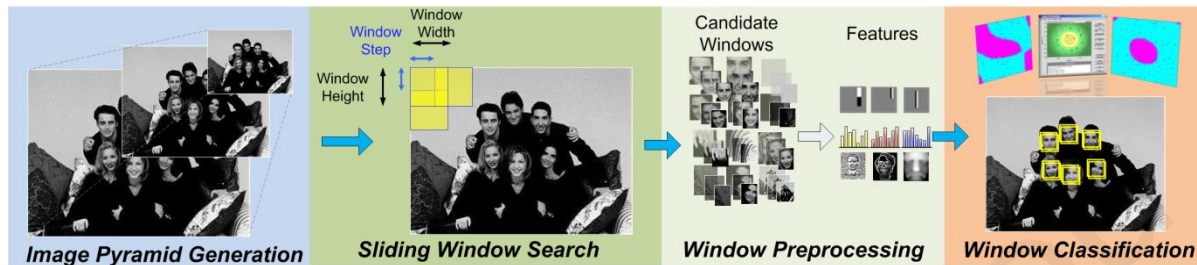


Figure 2-3. Object detection process

Object Detection Process: A) Image pyramid generation: The larger input image is downscaled. B) Sliding window search: A search window slides across the image every few pixels to extract smaller image regions. C) Window preprocessing: The image region is preprocessed to account variations and/or to feature extraction. D) Window classification: The image region is classified using a machine learning/pattern recognition algorithm.

2.3 Challenges and Trade-offs for Embedded Visual Object Detection

The two major challenges in meeting the performance metrics of intelligent embedded vision systems regard the complexity of the classification algorithm and the sliding window search process. A visual object detection system exhaustively scans all sub-windows in an image. Many downscaled images are produced from a single input image/video frame, each in turn producing a number of search windows, which increases the amount of data that must be processed by the classification algorithm. Evaluating all sub-windows becomes a tedious task when the classifier consists of several thousand training data/features and in addition features need to be extracted from the image.

2.3.1 Sliding Window Search Process

The overall search process can be a tedious task and impacts both processing speed which depends on the number of generated windows, as well as accuracy with regards to the granularity of the search. The parameters that impact the search process are: the search

window size, window pixel step, image size, and downsampling rate. The search window size is the smallest possible object size that can be detected and impacts the number of generated windows for a given frame size. The number of scales depends on the downsampling rate and is important since it enables detecting larger objects. The window pixel step is the distance between consecutive windows and affects the granularity of the detection system as it determines the percentage of the image region that will not be processed while also determining the number of windows for a given frame size. A small pixel step between windows ensures that the detection system will examine all possible candidates in the image, increasing the probability of finding all objects in the image. However, the number of generated search windows per scaled image version will increase and consequently decrease the frame-rate. Overall, these parameters are collectively used to find the number of windows generated from a single image frame. The number of generated windows can be calculated from the following:

$$\# \text{ of windows} = \sum_{n=0}^{\#scales-1} \left(\frac{IM_{width} - (W_{width} - W_{step})}{DR^n \cdot W_{step}} \right) \times \left(\frac{IM_{height} - (W_{height} - W_{step})}{DR^n \cdot W_{step}} \right) \quad \text{Equation 2-2}$$

where W_{width} is the window width, W_{height} is the window height, W_{step} is the window pixel step (assumed to be the same in both vertical and horizontal directions), IM_{width} is the input frame width, IM_{height} is the input frame height, DR is the downsampling rate, and n is the current iteration. The total number of iterations depends on the desired number of scales that need to be processed which can be down to the smallest window size. It is evident from the above equation that the number of windows is increased as the window pixel step becomes smaller, the downsampling rate decreases slowly (i.e. its value is closer to 1), the window size becomes smaller, or if many scales need to be searched. By appropriately setting the above four parameters for a given application it is possible to achieve an adequate trade-off between frame-rate and classification accuracy. The number of generated search windows can be the bottleneck in the frame-rate of an embedded object detection system. This has prompted research towards alternative search method that rely on color- and video- based approaches (e.g. skin detection, interest point detection, motion-detection, etc.) to reduce the search space. Although these methods reduce the locations where exploration needs to be performed they do

not provide a means to find the actual window size, as a result a scale exploration still needs to be performed.

2.3.2 Pattern Recognition Classification Algorithm

The computational demands for the implementation of a classification algorithm depend on the amount of training data and the complexity of the operations. Usually, the operations that need to be carried out may range from convolution of the input window with predefined kernels, to more complicated operations such as square roots and exponential functions. Complex operations take more time to compute and thus it may be necessary to approximate them or pre-store their values for fast rapid function evaluation. Of course this option also involves a trade-off between the loss due to the approximation and The amount of the resulting training data that may be required by the classification algorithm is directly related to the amount of training set and the algorithm complexity. The complexity of each algorithm, however, is different and depends on the training method and model parameters. Nevertheless, considering that appearance-based methods may need a large amount of data to achieve the targeted accuracy for visual object detection applications the underlying classification process is generally considered a tedious task.

2.3.3 Overview of Computing Platforms for Embedded Object Detection

This section aims to provide an overview of the different computing platforms that have been used in the design of vision systems [24] where real-time object detection application can be used in order for the reader to understand the capabilities and limitations of each platform in terms of detection speed performance, power dissipation as well as potential system size. These architectures are the multi-core Central Processing Unit (CPU), the Digital Signal Processor, the Graphics Processing Unit (GPU), and the Field Programmable Gate Array (FPGA). It must be noted, however, that the choice of platform also depends on the application scenario, algorithm, constraints, as well as cost and devoted development time.

A. Multi-core CPUs

Multi-core CPUs have been the trend in mainstream CPU architectures over the past decades in an attempt to find other ways to improve performance primarily due to three limiting factors (power consumption, memory latency and wiring delays, and limits of instruction-level parallelism) that have stalled the progression of single CPU architectures [25]. As the name suggests instead of having a single core with very high frequency and deep pipelines, it is replaced with two or more simpler cores that process different threads (a small sequence of program instructions). Multi-core CPU architectures have been the traditional processing platform for the implementation of computer vision algorithms since they offer high flexibility, ease of use, and fast development times. Recent multi-core systems consist of 4 – 8 physical CPU cores with operating frequencies in the order of GHz. They also have floating point support, and support for vector processing through single instruction-multiple data (SIMD) instructions. However, the parallelism and achievable performance is proportional to the number of cores, which are fewer compared to other platforms, setting a limit to the maximum achievable performance [24]. In addition, existing vision systems developed using high end CPUs that can provide real-time video rate require utilizing all the cores and thus exhibit high power consumption (the reader is referred to [26] for more details). As such, they are not suitable for embedded environments.. Reducing power consumption requires reducing the frequency or reducing the processing resources both of which end up reducing the performance. Hence, multi-core CPU systems have traditionally been used for security and surveillance systems on desktop computers where power consumption is not a key constraint, rather than embedded environments.

B. Digital Signal Processors (DSP)

A digital signal processor (DSP) is similar to a general purpose processor in the sense that it also has fixed logic, can execute a finite number of instruction types, and the instructions have a sequential flow [1]. However, the main distinction from general purpose CPUs is that their ISA is optimized for matrix operations, particularly multiplication and accumulation (MAC) which is the most common operation in signal processing applications, and expect a linear program flow with infrequent conditional statements where a large amount of data needs to be processed with the same mathematical program/operations. For this reason, they

provide SIMD (single instruction, multiple data) instructions to exploit parallelism by executing the same instruction on multiple data streams, as well as VLIW (Very Long Instruction Word) instructions to process different instructions with different data [1]. High performance DSPs often pack multiple processing cores with a general purpose CPU structure. They also include multiple DMA (Direct Memory Access) units and dedicated I/O units for fast access to off-chip memory. DSPs are an attractive platform for embedded vision systems, offering programmability, low-cost and low-power, and parallelism in the form of SIMD, VLIW, or both. However, the fixed number of processing resources on a DSP coupled with lower frequencies than multi-core CPUs, can be the bottleneck in the highest achievable performance [27].

C. Graphics Processing Units (GPUs)

Graphics processing units (GPUs) deal with processing of information in order to produce a 2D image from a scene (made up of known objects at different scales, orientations, size, distances, colours, shapes, etc.). This is the inverse to computer vision where the goal is to construct a description of a scene from an input 2D image. Recently GPUs have evolved from a dedicated graphics coprocessor to a host CPU, into a massively parallel programmable compute engine. GPUs employ grids of massively parallel programmable stream processing cores [28] that are efficient for high performance computing, while using additional fixed function hardware for graphics processing. Hence, since early 2002 – 2003 there has been a massive interest in utilizing GPUs for general purpose computing applications [29], called general-purpose computation on GPUs (GPGPU) [28], using C-style language definition and compilers such as NVIDIA's CUDA (which stands for Compute Unified Device Architecture).

The architecture of a GPU is drastically different from that of a CPU, in that transistors are used for computational processing units instead of caches and branch prediction and their architecture is optimized for high throughput instead of low latency. Consequently, GPUs offer order(s) of magnitude greater performance and are widely considered as the computational engine for the future. GPUs also have a different memory hierarchy which aims at delivering high bandwidth through wide memory busses and specialized graphics memory. In addition GPUs employ small read-only caches that help in reducing the bandwidth requirements on the main memory and benefit from small caches that capture spatial locality.

As the number of programmable processors increases significant amounts of on-chip storage are used to hold execution context, stream data, and temporary data. GPUs provide raw horsepower for compute intensive tasks requiring real-time performance such as in vision applications that. However, there are trade-offs to be made between power and performance as higher-end GPUs tend to have high power consumption (in the order of hundreds of watts [11]) [1]. Another potential drawback for use of GPU technology in embedded environments is the fact that even though programmability capabilities of GPUs have improved dramatically in the last few years, debugging and is still a challenging task [30]. Finally, necessary data transfers between GPU and the host CPU increase the latency of the application [30], something that is often not considered in embedded vision application implementations on GPUs. On the other hand, embedded GPUs (eGPUs) require have less demands in terms of power consumption but do not offer the same programmability features and have less processing and memory resources [12].

D. Field Programmable Gate Arrays (FPGAs)

Field programmable gate arrays (FPGAs) [31], as shown in Figure 2-4, are a type of reconfigurable integrated circuit made up of configurable logic blocks (CLBs) consisting of look-up tables (LUT) which are programmed to store given outputs for all input combinations, as well as other logic such as registers and logic gates. The CLBs are interconnected via an interconnection network of programmable switches and I/O Blocks are available on the perimeter to allow communication with external devices. In addition, modern FPGAs also come equipped with dedicated hardwired processing blocks such as embedded processors, MAC units, and embedded memory. All these components make FPGAs a fully programmable integrated circuit which the designer can use to develop any desired digital circuit while also taking advantage of the available dedicated resources.

A digital circuit (hardware architecture) is configured on the FPGA using a hardware description language (HDL) which specifies its behavior and functionality. The huge benefit of FPGAs compared to the aforementioned computing platforms is that the application circuit can be tailored to the demands of the application with respect to how the processing will be performed, the data flow, control and synchronization of various components and interfacing with I/O. Hence, FPGAs offer a large amount of parallel resources that can be exploited with

the appropriate hardware architecture in order to provide the necessary performance, and since they can be reconfigured also provide a high degree of flexibility. In addition, FPGAs offer a deterministic performance and a complete solution since a full system along with peripherals can be integrated on the same FPGA with a deterministic number of cycles needed to carry out the computations. However, the downside is that designing with FPGAs requires long development cycles which can be substantial, hence FPGAs are mostly used as prototyping platforms for potential future application specific accelerators for vision applications. In addition FPGAs often have lower frequencies than other platforms because of the complex fixed interconnection network which results in longer delay paths. However, the low frequencies can be compensated by designing a highly parallelized architecture. As such, there is a growing research effort to design dedicated hardware architectures for vision applications in order to provide the real-time performance along with the lower power consumption of FPGAs.

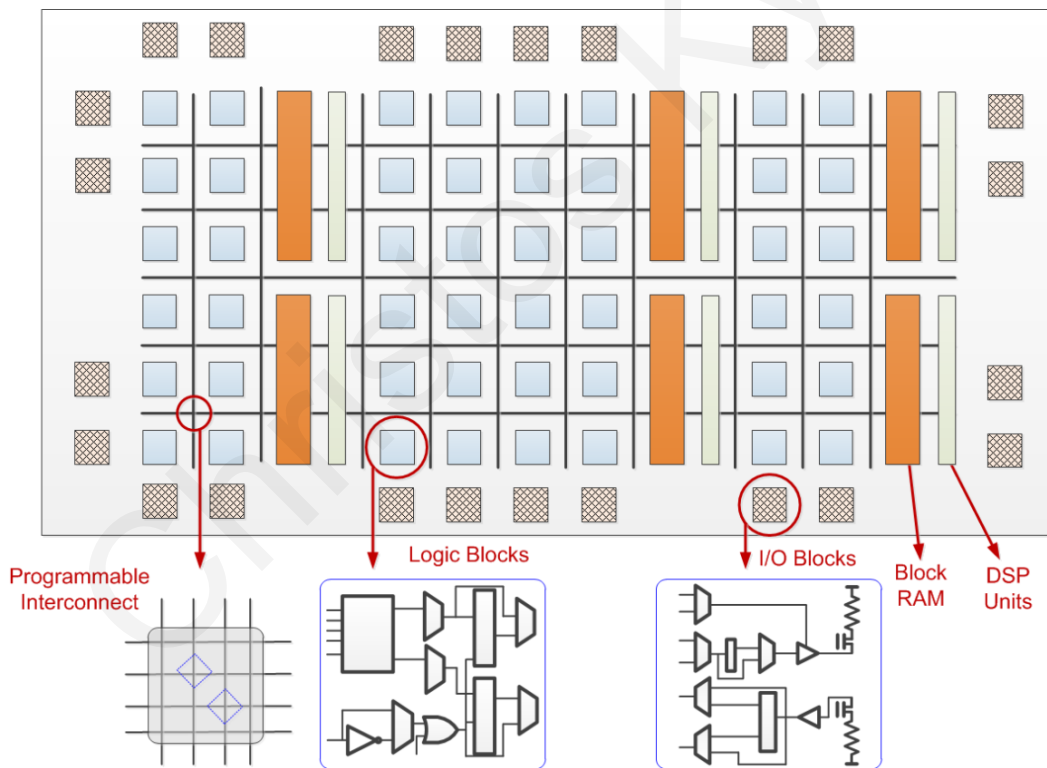


Figure 2-4. Architecture of a Field Programmable Gate Array (FPGA)

E. Towards a Heterogeneous Embedded Vision System-on-Chip

The above analysis of the main concepts and properties of each computing platform demonstrates that each platform comes with its own strengths and weaknesses. As such, given the diverse nature of computer vision applications in terms of data flow and operations and the need for specialization as well as flexibility it is reasonable to envision a heterogeneous embedded vision system-on-chip (SoC) [24],[32] where the above processing platforms will be utilized to carry out different application tasks (Figure 2-5).

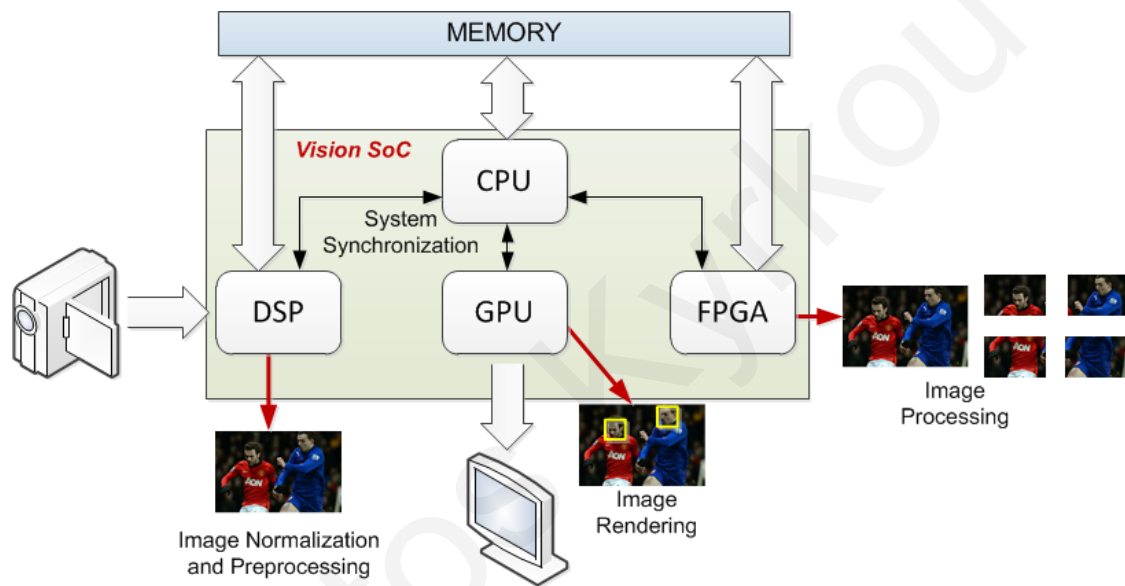


Figure 2-5. A heterogeneous embedded vision SoC

For example, for a vision processing application the control flow mechanisms of a CPU are well suited for system supervision and synchronization purposes, the signal processing capabilities of a DSP can be used to handle video transmission and reception, a GPU can be used to handle image preprocessing, manipulation and rendering, and an FPGA can be used to provide customized and optimized application specific parallel processing for image analysis. Anticipating this trend towards a heterogeneous platform [33] this thesis addresses the design of a hardware architecture for the acceleration of object detection systems that can be used for real-time embedded applications. FPGAs are used as a prototype to evaluate the architectures, however, they can be used with different programmable hardware platforms. Compact low-power and real-time architectures can be used to design specialized processors which can be

used to embed visual detection capabilities in cameras, handheld devices, autonomous vehicles and humanoid robots. However, there are various design challenges associated with the development of hardware architectures which are summarized next.

- *Hardware Design Issues and Challenges*

Designing a hardware architecture for object detection that can be used either with FPGAs or with other hardware platforms, involve a trade-off between different factors that affect both detection accuracy and performance. This are briefly mentioned below but will become more apparent in the following sections and chapters.

- A. *Input & Training Data Representation*

The data representation plays a central role in the implementation of a hardware architecture for object detection. Typically, for object detection applications which concern pixel images the input is represented by 8-bits for grayscale images (intensity values of 0 – 255), and 24 bits (8-bits per color channel) for RGB (red-green-blue) images. The vast majority of object detection applications perform classification on the grayscale images. However, it is not necessary to use 8-bits since the input can be preprocessed and normalized to fit the requirements of the hardware implementation platform. As such, depending on the preprocessing method and feature extraction approaches bits necessary to represent the value can vary. In addition, the value may not be an integer such as in grayscale images but a real number. Training data (weight and threshold values) are also typically real numbers.

The representation of real-numbers in hardware can be done either floating or fixed point arithmetic and twos complement or signed representation. Usually fixed point twos-complement representation is used due to the simpler hardware needed for mathematical operations. However, there needs to be a design space exploration to find the optimal number of bits to use in the representation in order to both minimize the memory requirements and preserve the targeted accuracy rate.

- B. *Parallel Processing*

A pattern recognition algorithm requires processing each input data with training data in order to obtain the classification output. The amount of training data that needs to be processed by the algorithm and the total amount of input data that need to be processed per

input image affect the performance. Hence, in order to provide real-time performance the hardware architecture needs to be designed in such a way as to facilitate parallel data processing. There are two possible ways to exploit parallelism in object detection systems. The first is pixel-level parallelism which aims at processing the data of a single window in parallel with one or more training data. The second is window-level parallelism which processes multiple windows with one or more training data, possibly sacrificing some speed for the classification of one window. A combination of window- and pixel-level parallelism is also possible but is not as simple and requires a more rigorous design-space exploration. Of course an important factor to consider is the available resources and how the data-flow between them can facilitate each level of parallelism. Assuming that the hardware resources are available the limiting factor is then how fast can the architecture be loaded with data. As such, providing an efficient data flow is equally important to having a parallel architecture. Furthermore, the complexity of controlling and managing a more parallel system is also an important aspect to consider.

C. Memory and I/O

The memory issues concerning an object detection system include latency, access, size, structure, and bandwidth. External memories such as DRAM or compact flash are used to store image/video frames. An image buffer can be used as local memory to store the active image frame on-chip to speed up the classification procedure, and provide more flexibility in the way the image memory is accessed. Memory access is important to provide a higher degree of parallelism. Additionally, image buffer structures can be tailored to accommodate the system requirements (e.g. by distributing the image into banks to provide parallel access). In many cases where on-chip memory prohibits the storage of a whole image frame, a window buffer is used to store only the window that needs to be processed.

In addition to the image data storage requirements the pattern recognition algorithm also requires the storage of training data used for classification. The frequency with which the training data is accessed affects how the complicated the memory management becomes. If the request to training data is sparse, then it is preferable to store the training data in an off-chip memory, and fetch only the currently required data. Consequently, on-chip memory resources

can then be used for the image frames. However, if the access to training data is frequent then using on-chip memory to provide fast and parallel access is the preferred option.

2.3.4 Applications of Visual Object Detection Systems

Thus far this section has provided the basic concepts for visual object detection from an overview of the detection process to the available computing platforms. As a final note it is important to consider how a hardware accelerated visual object detection system can be used and what are the targeted embedded applications. As mentioned earlier in the text, real-time performance for video applications is considered to be around 30 – *FPS*. This number stems from how humans perceive motion and is the target for human-centric applications which require real-time interaction and responsiveness. As such, for those applications DSP processing platforms, and when the power budget permits it CPUs and GPUs, can provide the necessary performance of around 25 – 30 *FPS*. However, it is not difficult to envision other scenarios where the system response and processing rates need to be higher than this value. For example, applications such as cloud image sequence analysis, quality control in industrial environments require detection to be performed at higher frame-rates, while other applications such as surveillance from multiple cameras, automated driver assistance, smart transportation systems, and aerial and terrestrial autonomous navigation, also require low-power consumption. The higher performance is necessary in order to alert or intervene with a host within given time limits, or in the case of multiple cameras where the input stream rate increases with each image sensor, or even when additional tasks need to be implemented such as recognition and analysis. These cases require well over 30 *FPS* [9], while also operating under relatively small power budget for the whole system [1],[10]. These are the types of applications that the research in this thesis attempts to tackle by demonstrating how customized hardware architectures can be used to accelerate object detection applications and provide high frame-rates and low-power consumption using FPGAs as the prototyping and evaluation platform.

A description of the background theory and details for two widely used classification algorithms for visual object detection which are considered in this thesis, namely the Adaboost-based Haar-cascade classifier, and Support Vector Machines, are provided in the

following sections. In addition, through this overview the trade-offs and challenges in the design of these classification algorithms will become more apparent.

2.4 Viola and Jones Framework for Rapid Object Detection

The following sections detail the Viola and Jones detection framework [15] which incorporates a set of methods and approaches that result in rapid object detection. It was the first object detection framework to provide competitive accuracy rates with a real-time processing rate of 15 FPS back in 2001. It can be used for a variety of objects but was primarily motivated by face detection. It was implemented as a part of Intel's OpenCV [18] computer vision library. There are three main components in the framework, integral image representation and evaluation of rectangular features, feature selection and learning through AdaBoost, and cascade classification architecture. All concepts are described next in more detail.

2.4.1 AdaBoost Boosting Algorithm

Boosting is a general method for improving the accuracy of any given learning algorithm. There are many variants of boosting methods which mainly differ in the training process. One of the most well known boosting approaches is called Adaptive Boosting (AdaBoost) [34]. The main advantage of AdaBoost is that it does not suffer from overfitting. It is based on the idea that a single strong classifier can be created from a set of weak classifiers. A weak classifier can be considered as a classifier which manages to label examples only slightly better than random guessing. In contrast, a strong learner is a classifier which manages to correlate arbitrarily well with the true classification outcome. Boosting then tries to produce accurate classification prediction rules (strong classifiers) by combining not so accurate, rough, rule-of-thumb classifiers (weak classifiers). Specifically, the strong classifier H is a weighted sum of all the weak features h . So for a new unknown input the outcome of weak classifiers h_i are multiplied with their respective weight values a_t derived from the training phase to predict the class the class. The main steps of the AdaBoost boosting algorithm are outlined in Figure 2-6.

-
-
- Given example data $(x_1, y_1) \dots (x_n, y_n)$ where $y_i = \{-1, 1\}$ for negative and positive examples respectively.
 - Initialize weights w_i for all samples so that they form a probability distribution
 $\rightarrow w_i = \frac{1}{n}, i = 1, \dots, n$
 - For $t = 1, \dots, N$
 - i. Find classification hypothesis h_j which minimizes the error
 - Calculate error ϵ_j of h_j for each example with respect to w_t
 $\epsilon_j = \sum_{i=1}^n w_t |h_j(x_i) - y_i|$
 - Choose the classifier h_t , with the lowest error ϵ_t .
 - If $\epsilon_t > \frac{1}{2}$ stop. Set $T=t-1$.
 - ii. Update the weights for next round: $w_{t+1,i} = \frac{w_{t,i}}{Z} \times \beta_t^{1-e_i}$, where $e_i = 0$ if example x_i is classified correctly, $e_i = 1$ otherwise, and $\beta_t = \frac{\epsilon_t}{1-\epsilon_t}$, where Z is a normalization factor such that the group of w form a probability distribution
 - iii. If $\epsilon_t \geq 0.5$ and set $T = t - 1$ and stop
 - iv. Find total strong classifier error CE :
 - $CE = \sum_{m=1}^t E_m(t, a_m, h_m(x))$,
where the current classifier error is $E_m = \frac{1}{n} I \sum_{k=1}^n I(t, a_m, h_m(x_k))$
and $I(\cdot) \begin{cases} 0, & \text{if } x_k \text{ is correctly classified} \\ 1, & \text{if } x_k \text{ is not correctly classified} \end{cases}$
and $a_m = \frac{1}{2} \log \left(\frac{1}{\beta_m} \right)$
 - Set $T=t$
 - if $CE = 0$ stop
 - The final strong classifier with T stages is:
 $H(x) = \text{sign}(\sum_{t=1}^T a_t h_t(x))$, where $a_t = \log \frac{1}{\beta_t}$.

Figure 2-6. Outline of the Adaptive Boosting (AdaBoost) algorithm

At each iteration t of the AdaBoost algorithm, a weak classifier is trained to produce a hypothesis h_j that classifies the training samples x_i . The error of this hypothesis with respect to the current weight values is calculated as the sum of the errors of the instances misclassified by h_j . AdaBoost requires that this error be less than $1/2$ so that the classifier effectively performs slightly better than random guessing. If a classifier cannot meet this requirement, the algorithm aborts and the current structure of weak classifiers is chosen as the strong classifier H_j . Otherwise, the normalized error β_t , is then computed so that the actual error that is in the $[0, 0.5]$ interval is mapped to $[0, 1]$ interval. The normalized error is used in the weight update

rule. Then the weights are normalized so that they correspond to a probability distribution. The way that the weights are updated every iteration ensures that subsequent weak classifiers focus on increasingly difficult instances as the weights of the instances that are misclassified are effectively increased. This update rule ensures that the weights of all correctly classified instances and the weights of all misclassified instances always add up to 1/2. More specifically, the requirement for the training error of the base classifier to be less than 1/2 forces the algorithm to correct at least one mistake made by the previous base model. Once the training is complete, samples are classified by an ensemble of T weak classifiers using weighted majority voting, where each classifier h_i receives a voting weight that is inversely proportional to its normalized error. The weighted majority voting then chooses the class that receives the highest total vote from all classifiers. This process trains a single strong classifier H_j . The next strong classifier H_{j+1} is trained by first filtering out the samples that are correctly classified by the current strong classifier structure H_1 to H_j as described in Section 2.1.2.

2.4.2 Haar-Features and Integral Image Representation

Viola and Jones introduced a set of simple features for computer vision applications derived from Haar wavelets [35], instead of directly using pixel intensity information. These features, called *Haar – like features* due to their intuitive similarity with Haar-wavelets, act as filters that can detect the presence or absence of certain visual characteristics in an image (lines, corners, etc). Each feature consists of a set of black and white rectangles in a horizontal or vertical structure (Figure 2-7). A feature can have 2-4 rectangles in different configurations as shown in Figure 2-7 and so each can detect certain image features and object patterns. The computation of Haar-like features involves computing sums of image regions and finding their difference. Specifically it is done by overlaying the feature over the input image, as in a convolution operation, and calculating the sum of the pixel values in the white rectangles of the feature, minus the sum of pixel values in the black rectangles, (Figure 2-7). The resulting value of the difference of white and black rectangles can indicate the presence of visual features in the image. Thus a group of Haar features can potentially describe an object and its characteristics and can be used to categorize images.

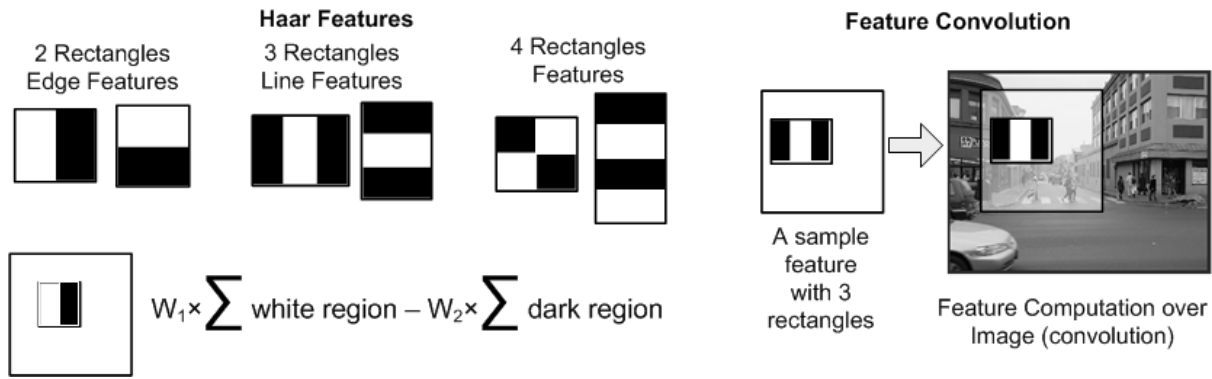


Figure 2-7. Examples of Haar-like features

A potential drawback of Haar-features, however, lies in the fact that for each feature, multiple rectangle sums need to be computed in order to speed up the feature computation, Viola and Jones proposed an alternative input image representation, called the *integral image*. The integral image is simply a transformation of the original input image, to an image where each pixel location holds the sum of all the pixels to the left and above of that location [15], as shown in Figure 2-8. The advantage of using the integral image is the ability to compute the sum of a rectangle in constant time. As shown in Figure 2-8 (rectangle computation), the computation for rectangle D is simply two additions and two subtractions of the four corner points of the rectangle when using the integral image rather than the original image. Hence, regardless of the feature or search window size, only four values per rectangle are necessary to compute the value of each feature. Additionally, the location of the rectangles within each feature is predetermined from the training set, hence to evaluate each rectangle the offsets d_x and d_y values from the starting coordinate of each feature are needed (Figure 2-8). The offset coordinates are part of the training set, where each feature is associated with a list of the feature's rectangles and the four pairs of (d_x, d_y) offsets necessary. This holds true during feature upscaling as well, as since the rectangle coordinates are fixed, d_x and d_y are also scaled linearly with the scale factor. For example, if a feature scales from 24×24 to 30×30 , (i.e. a scale factor of 1.25) a rectangle that in the initial feature would be located at starting coordinate $(d_x = 4, d_y = 8)$ would be mapped to $(d_x = 5, d_y = 10)$, with d_x and d_y multiplied by the scale factor (rounded to the nearest integer).

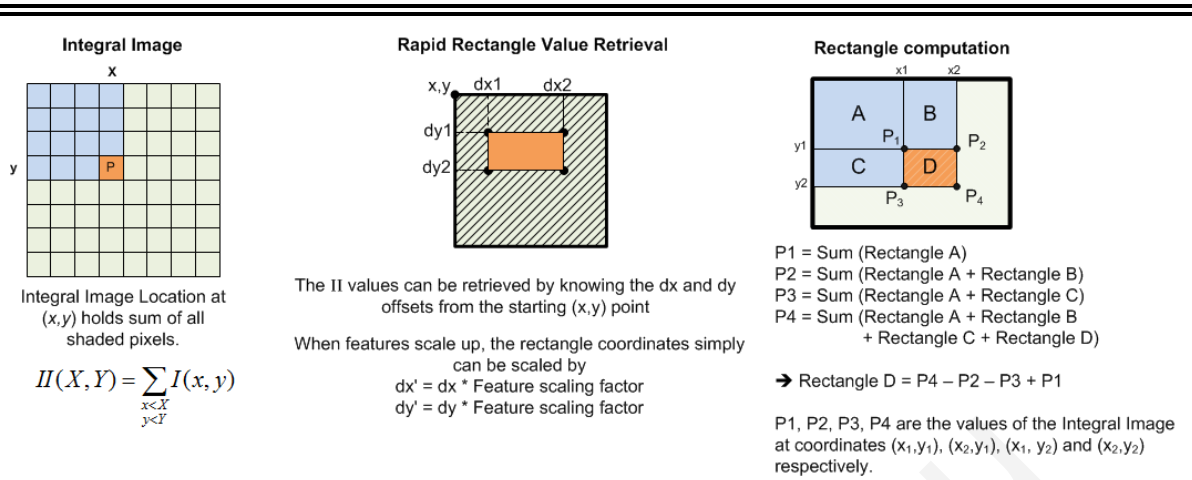


Figure 2-8. Integral image and fast feature computation

2.4.3 Haar-Feature AdaBoost-Based Cascade Classifier

The algorithms and concepts described in the previous sections have been incorporated into a dedicated framework for robust real-time object detection by Viola and Jones [15]. Specifically AdaBoost [36] was used as part of a learning algorithm to select a number of Haar-features from a larger pool, in order to construct a cascade classifier of strong and accurate classifiers from the weaker Haar classifiers. The combined framework presents two significant advantages that have contributed to it being considered state of art approach for object detection. First, it has the ability to quickly eliminate non-object regions, and second the classification process itself and the computations per window is simple and fast. A group of features composes a strong classifier, the outcome of which is the sum of the feature outcomes. Multiple strong classifiers with a varying number and type of features are arranged in a cascade of stages. The sum of all weighted feature rectangles in the Haar-like feature is used to determine the feature sum, if this sum exceeds a certain value then it is set to a predetermined value obtained from the training set, otherwise it is set to another predetermined value, also obtained from the training set. All feature sums are computed and accumulated to compute the stage sum. At the end of each stage, the stage sum is compared to a predetermined *stage threshold value* (t_o). If it is larger, the image is a successful candidate region to contain the object of interest, otherwise, it is discarded, and omitted from the rest of the computation. This technique accelerates the process of rejecting an image

region that does not contain objects of interest, so that computation time will focus only on successful candidates. The original algorithm [15] used features starting at 24×24 pixels, however, the feature size can vary with the application. The number of rectangles for each feature and their configuration vary also depending on the object of interest and the more informative visual features of that object.

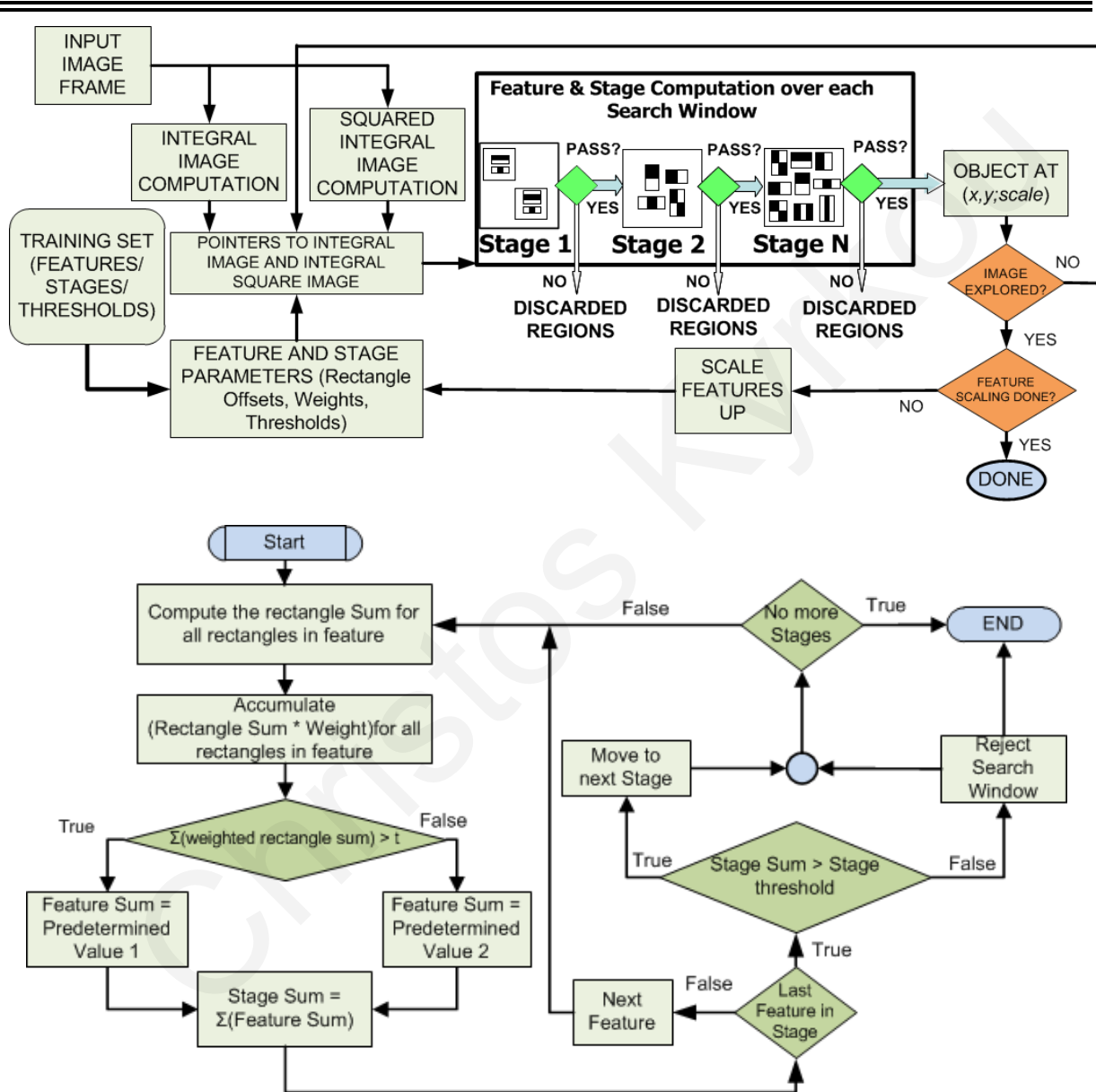


Figure 2-9. Viola and Jones algorithm flow

(top) Outline of the Viola and Jones detection procedure. (bottom) Stage evaluation outline.

Objects in the image frame which are larger than the search window and the feature, do not get detected. This is usually solved by downscaling the original image frame, subsequently reducing the object size, and making it detectable. However, Viola and Jones suggest enlarging the feature instead, this way, image data that could potentially be lost by downscaling remains, and the features that are simply black and white rectangles, scale linearly without loss of data. Consequently, when all search windows finish, features are scaled by a predetermined factor, to detect objects larger than the original feature size. The computation is repeated again, using new training values set for the larger scale, until all objects of all sizes are detected in the image frame and until the size of the feature reaches the size of the largest possible object (in terms of pixels) in the input image frame. The amount of scaling also impacts the detection speed significantly, which further stresses the need for rapid feature computation. Figure 2-9 shows the overall detection process and classification algorithm flow chart and computation stages.

$$VAR = \sum_{i=0}^{\# \text{ of pixels}} \left[\frac{\sum_{i=0}^{\# \text{ of pixels}} X_i}{AREA} \right]^2 \text{ and } \sigma = \sqrt{VAR} \quad \text{Equation 2-3}$$

$$t = t_o * \sigma_i \quad \text{Equation 2-4}$$

All the above computations are essential for the classification process required by the detection algorithm. There are also some additional computations necessary to deal with the varying characteristics in which the object of interest may appear, due to the lighting and environmental variations. These computations regard a lighting correction technique to compensate for these variations. This technique requires the computation of the squared integral image for each input image (each image location holds the sum of the squared pixel values). This is necessary to compute the variance (VAR) and the standard deviation (σ) of the image, to compensate for the lighting variations, as shown in Equation 2-3. The standard deviation is multiplied with the original feature threshold (t_o) given in the training set, to obtain the *compensated threshold* (t) (Equation 2-4) which dynamically takes care of any lighting variations encountered during the detection stage, and improves the overall accuracy of the algorithm. It is needed to be done only once for every search window however, all

subsequent features evaluated over that search window can use the computed standard deviation value as shown in Equation 2-3.

2.4.4 Parallelism Opportunities and Computational Trade-offs

Overall, the Viola-Jones detection framework offers a lot of opportunities to exploit parallelism. First, each downsampled image can be processed independently from the other versions. Second, within each image the sub-windows can all be classified in parallel. Third each feature in a stage can be processed in parallel and independent from each other. Finally, the operations for computing the rectangle sums within the feature can be done together. Of course these parallel operations require that the data can be accessed in parallel. Also the fact that the computations of the rectangles require only four values once the integral image is computed makes the computation and access to the integral image values all the more important. Furthermore, computing the integral image for every window is quite tedious both in terms of memory access and computations. Hence, there is the need to not only compute the integral image only once and fast, but to also exploit the memory hierarchy and local data storage and access in order to achieve high performance. Also the scaling method is important. As the features are scaled up the access to integral image becomes sparser hence making it difficult to exploit spatial localities for fast data access. On the other hand, scaling the image up instead of the features requires re-computing the integral image for every scale and may also result in some accuracy loss.

2.5 Support Vector Machines Overview

The goal of this section is to provide the central ideas of Support Vector Machine learning and overview of the basic concepts. The reader is referred to textbooks such as [37] and [38] for a more detailed and in-depth look. Support Vector Machines (SVMs) have been widely adopted since their introduction by Cortes and Vapnik [16], [39]. They are considered one of the most powerful classification engines due to their mathematical background that is based on statistical learning and is able to accurately model classification boundaries [39]. Consequently, there has been growing interest in utilizing SVMs in numerous applications including visual object detection systems [40].

2.5.1 Support Vector Machine Formulation - Training

A SVM tries to find the mapping function f for binary classification problems (where class label y is either -1 or 1) by utilizing some basic concepts from *Maximum – Margin – Classifiers* and *Kernel Functions*. The SVM algorithm is derived by applying *Lagrangian optimization theory* to the *Maximum – Margin – Classifiers* optimization problem [37]. Training SVMs requires solving a *Constrained Quadratic optimization Problem* (CQP). An overview of these fundamental concepts of SVMs follows next.

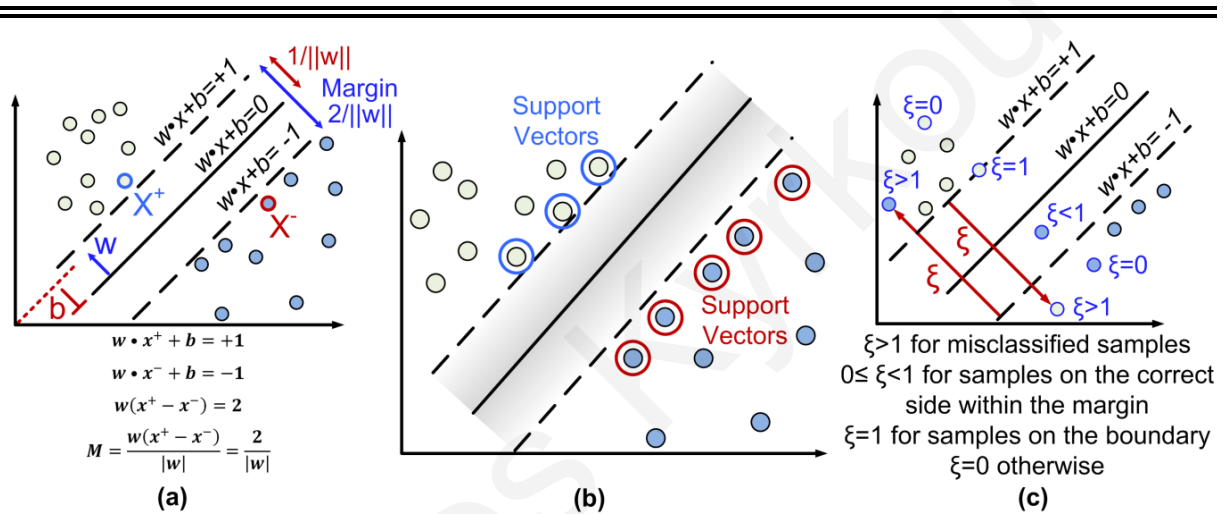


Figure 2-10. Support vector machine main concepts

(a) Maximum margin and supporting hyperplanes (b) Slack variables (c) Support vectors

A. Maximum Margin Classifiers

Maximum margin classifiers are linear classifiers which given a two class data set try to construct the "best" hyperplane that separates the two classes. The best hyperplane is found in terms of the boundary from each class. Hence, in order to find the best separating hyperplane the distance between the hyperplanes which lie at the boundary of each class needs to be maximized [19]. The hyperplane can be described by a *normal vector* w and an offset *bias* term b . The equation of the hyperplanes at the boundary of each class is given by Equation 2-5 where x_i is a data point and y_i its corresponding class label. The distance between these two hyperplanes, as shown in Figure 2-10, is the margin between the two classes and is equal to $\frac{2}{\|w\|_2}$. Thus in order to maximize the margin it is necessary to minimize the normal vector

norm. This results in the optimization problem for maximum-margin hyperplane classifiers shown in Equation 2-6. Those points for which the equality in Equation 2-6 holds lie on the supporting hyperplanes (Figure 2-10) and referred to as *support vectors (SVs)*, and their removal from the data set would result in a change in solution of the optimization problem.

$$\begin{aligned} (a) \quad & w \cdot x_i + b \geq y_i, \text{ for hyperplanes at the class boundaries} \\ (b) \quad & w \cdot x_i + b = 0, \text{ for the separating hyperplane} \end{aligned} \quad \text{Equation 2-5}$$

$$\min_w \left(\frac{\|w\|^2}{2} \right) \text{ such that } y_i(w \cdot x_i + b) \geq 1 \quad \forall_i \quad \text{Equation 2-6}$$

However, the above optimization problem can find the best separating hyperplane given that the problem under consideration is linearly separable and that all samples lie in the "correct side" of the hyperplanes. Such problems are not often encountered in real-world scenarios due to outliers/noise. An extension of the maximum margin hyperplane called the *soft margin* can allow for some data samples to be misclassified to deal with such problems. The concept of soft margin permits for the constraints on some of the training data to be relaxed by adding slack variables ξ_i to the constraint optimization problem. However, this may lead to trivial solutions where all the data samples are considered as noise/outliers and are thus misclassified. To prevent such cases the slack variables are also added to the minimization function of the optimization problem in the form of a penalty factor. The penalty of misclassification is controlled by a regularization parameter C , which effectively controls the number of misclassifications.

The formulation thus far assumes that the data samples of a given set can be adequately separated by a linear separating hyperplane even in the presence of a few outliers. However, many data sets are not linearly separable in which case a linear function will not result in good data separation. Hence, the SVM formulation needs to be enhanced to account for non-linear separations of the data. To this end, a mapping function can be used to project the data from the input space to a feature space of a higher dimensionality, so that data become more easily separated or better structured. Doing so makes it possible to compute a linear separating hyperplane for the data that corresponds to non-linear separation in the original space. Such a mapping function φ maps points from an n -dimensional space to an m -dimensional space

where $m > n$, then by replacing x_i with $\varphi(x_i)$, the normal vector w can be found in \mathbb{R}^m . Taking everything into consideration the SVM optimization problem then becomes:

$$\min_w \left(\frac{\|w\|^2}{2} + C \sum_i \xi_i \right) \text{ such that } y_i(w \cdot \varphi(x_i) + b) \geq 1, \xi_i \geq 0 \forall_i \quad \text{Equation 2-7}$$

B. Lagrangian Optimization

The above optimization problem is a quadratic programming (QP) problem which requires minimizing a quadratic function subject to linear constraints on the problem variables. This problem can be solved using its dual representation which is always convex and thus ensures a bound on the objective value and also is in general easier to compute than the original problem. To obtain the dual representation of the QP optimization problem in Equation 2-7, one first needs to form the Lagrangian equation using nonnegative Lagrange multipliers and add them to the constraints of the objective function. In general for problems where we need to find local minima and maxima of a function subject to equality constraints of the form $\min F(x, y) \text{ s.t. } g(x, y) = c$, the general form of the Lagrangian is given by $L(x, y, a) = F(x, y) + a(g(x, y) - c)$, where $F(x, y)$ is the cost function and $g(x, y)$ are the linear constraints. Hence, for the specific problem at hand the Lagrangian is given by Equation 2-8. The Lagrangian now needs to be optimized with respect to the variables a, b , and w . However, in order for the solution to this problem to be optimum some conditions need to be met. These are called Karush-Kuhn-Tucker (KKT) conditions [38] (Figure 2-11) and allow using the Lagrange multipliers formulation when the constraints also have inequalities instead of only equalities.

-
-
- All a 's need to satisfy the following within a small epsilon ε , typically 10^{-3} and u_i is classification result
 - i. $a_i = 0 \Leftrightarrow y_i u_i > 1$ An a is 0 iff an example is correctly labeled with room to spare
 - ii. $a_i = C \Leftrightarrow y_i u_i < 1$ An a is C iff an example is incorrectly labeled or in the margin area
 - iii. $0 < a_i < C \Leftrightarrow y_i u_i = 1$ An a is properly between 0 and C (is "unbound") iff that example is correctly labeled and lies on the boundary correctly

Figure 2-11. Karush-Kuhn-Tucker (KKT) Conditions

The KKT conditions are formed by taking the derivative of the Lagrangian with respect to the three aforementioned variables thus formulating Equation 2-9-Equation 2-11 as shown below. The KKT conditions provide relationships for parameters w , a , and b and are necessary in order to find a solution to the optimization problem. The solution for the vector w can be computed using Equation 2-10, which, however, implies that the corresponding alpha coefficients (also called Lagrangian multipliers) a_i , need to be found. Substituting these relationships into the Lagrangian equation results in what is known as the *dual* optimization problem shown in Equation 2-12, that needs to be solved in order to obtain a solution and which now is only expressed in terms of the Lagrange multipliers and dot-products of the projected data samples $\varphi(x)$.

$$L(x, y, a, b, \xi, \lambda) = \frac{\|w\|^2}{2} + C \sum_i \xi_i + \sum_i \alpha_i (y_i (w \cdot \varphi(x_i)) + b) + 1 - \xi_i - \sum_i \lambda_i \xi_i$$

$$\Rightarrow \frac{\|w\|^2}{2} + C \sum_i \xi_i + \sum_i \alpha_i y_i (w \cdot \varphi(x_i)) + \sum_i \alpha_i y_i b + \sum_i \alpha_i - \sum_i \alpha_i \xi_i - \sum_i \lambda_i \xi_i$$

Equation 2-8

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_i \alpha_i y_i = 0$$

Equation 2-9

$$\frac{\partial L}{\partial w} = 0 \Rightarrow w = \sum_i \alpha_i y_i \varphi(x_i)$$

Equation 2-10

$$\frac{\partial L}{\partial \xi} = 0 \Rightarrow \alpha_i = C - \lambda_i$$

Equation 2-11

$$\max_a \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \varphi(x_i) \varphi(x_j) \text{ s.t. } \sum_i \alpha_i y_i = 0, 0 \leq \alpha_i \leq C, \forall_i$$

Equation 2-12

C. Kernel Trick

The final part in the formulation of SVMs concerns the computation of the mapping function φ . In some cases the mapping functions may not be known explicitly and even if it is known it may be computationally costly to calculate it for each sample. The *kernel trick* is a mathematical tool [41] which can be applied to any linear algorithm, where the data appear in the form of dot/inner products $\langle x_i \cdot x_j \rangle$, to transform it into a non-linear one. Wherever a dot product is used, it is replaced by an appropriate *kernel function* such that $k(x_i, x_j) = \varphi(x_i) \varphi(x_j)$. Because kernels are used, the mapping function φ , which in cases may be infinite

dimensional and hence infeasible to compute, does not need to be ever explicitly computed and the mapping is done implicitly using kernels (Figure 2-12).

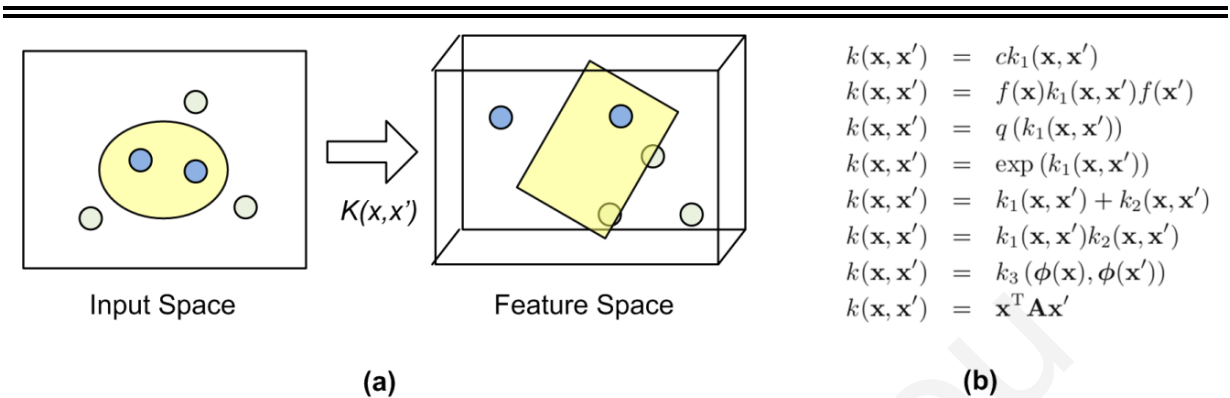


Figure 2-12. Kernel trick and kernel construction

(a) Kernel trick illustration. (b) Given valid kernels $K_1(x, x')$ and $K_2(x, x')$ the illustrated kernels will also be valid. In which $c > 0$, is a constant, $f(\cdot)$ is a polynomial with nonnegative coefficients, $\phi(x)$ is a function x to \mathbb{R}^M , $K_3(\cdot, \cdot)$ is a valid kernel in \mathbb{R}^M , and \mathbf{A} is a symmetric positive semi-definite matrix.

When using the kernel trick there are no constraints on the form of the mapping, which could even lead to infinite-dimensional spaces, nor to the nature of the input vectors, since dot products could be defined between any kind of structure, such as trees or strings. In addition kernel functions can be interpreted as a similarity measure of two vectors in the input space.

However, Kernel functions must be continuous, symmetric, and most preferably should have a positive (semi-) definite Gram matrix [42]. Kernels which are said to satisfy the Mercer's theorem [19] are positive semi-definite, meaning their kernel matrices have no non-negative Eigen values. The use of a positive definite kernel insures that the optimization problem will be convex and thus it has a unique solution. However, many kernel functions [43] which are not strictly positive definite (i.e. are only positive semi-definite for certain parameter alues) also have been shown to perform very well in practice (shown in Table 2-1). An example is the hyperbolic tangent or sigmoid kernel Equation 2-17, which is not positive semi-definite for certain values of its parameters. Existing kernel functions can be used to construct new kernels [44] as shown in Figure 2-12. Choosing the most appropriate kernel for a given application is a difficult task and often depends highly on the problem at hand and fine tuning its parameters can easily become a tedious and cumbersome task. Common kernel functions used in SVMs

are shown in Equation 2-13 through to Equation 2-20, in which a_i are the two vectors and *const, degree, sigma, p, theta, beta, and gamma* are kernel-specific parameters that define the behavior of each kernel.

TABLE 2-1 SUPPORT VECTOR MACHINE KERNEL FUNCTIONS

Standard Kernels	
<p>Linear: $K(x_i, x_j) = (x_i \cdot x_j)$</p> <p>The Linear kernel is the simplest kernel function. It is essentially a dot product and the feature space is simply the same as the input space.</p>	<p>Equation 2-13</p>
<p>Polynomial: $K(x_i, x_j) = ((x_i \cdot x_j) + \text{const})^{\text{degree}}, \text{const} \geq 0, \text{degree} \in \mathbb{N}$</p>	<p>Equation 2-14</p>
<p>Homogeneous Polynomial: $K(x_i, x_j) = ((x_i \cdot x_j))^{\text{degree}}, \text{degree} \in \mathbb{N}$</p> <p>Polynomial kernels are well suited for problems where all the training data is normalized. They allow to model feature conjunctions up to the order of the polynomial. Homogeneous 2nd degree polynomials allow for exact solution to the reduced support vector set problem.</p>	<p>Equation 2-15</p>
<p>RBF: $K(x_i, x_j) = \exp\left(-\frac{\ x_i - x_j\ ^2}{2 \times \text{sigma}^2}\right), \text{sigma} > 0$</p> <p>The adjustable parameter sigma plays a major role in the performance of the kernel and the resulting generalization capabilities. This kernel allows to pick out circles or hyperspheres.</p>	<p>Equation 2-16</p>
Other Kernels (Conditionally Positive Semi Definite)	
<p>Hyperbolic Tangent: $K(x_i, x_j) = \tanh(p(x_i \cdot x_j) + \theta), a > 0, c < 0$</p> <p>An SVM model using the hyperbolic tangent kernel function is similar to a two-layer neural network. Even though this kernel is only conditionally positive definite, it has been found to perform well in practice.</p>	<p>Equation 2-17</p>
<p>Log Kernel: $K(x_i, x_j) = -\log\left(\ x_i - x_j\ ^\beta + 1\right), 0 < \beta \leq 2$</p>	<p>Equation 2-18</p>
<p>Power Kernel: $K(x_i, x_j) = -\ x_i - x_j\ ^\beta, 0 < \beta \leq 2$</p> <p>The log and power kernels seem to be particularly interesting for images, however, they are only conditionally, positive definite.</p>	<p>Equation 2-19</p>
<p>Hardware Friendly Kernel: $K(x_i, x_j) = 2^{-\gamma\ x_i - x_j\ _1}, \gamma > 0$</p> <p>Approximation of the RBF kernel. When used in conjunction with CORDIC algorithms it can be implement in hardware without multipliers.</p>	<p>Equation 2-20</p>

Incorporating the kernel trick into Equation 2-12 yields the final optimization problem (Equation 2-21) that needs to be solved to find the solution for the SVM parameters. The solution to this problem can be found by using QP solvers which attempt to globally solve the optimization problem by considering all the data samples together. However, this requires a lot of memory and is a slow process proportional to the size of the training set. Instead iterative methods can be used such as the very popular and widely used Sequential Minimal Optimization (SMO) [45] algorithm found in popular SVM training packages such as MATLAB [46], and LIBSVM [44]. These algorithms decompose the problem into a set of smaller problems which are then solved analytically, thus improving the efficiency of the training process, since they require no extra matrix storage, while staying close to the optimal solution [40]. Specifically, the SMO algorithm attempts to solve the smallest possible sub-problem at every step which is to find the optimal values of two Lagrange multipliers (a_i). The basic steps of the SMO [45], also shown in Figure 2-13, can be summarized as follows: *i*) Choose two Lagrange multipliers to jointly optimize *ii*) Find the optimal values for Lagrange multipliers and *iii*) Update the SVM to reflect the new optimal values.

$$\max_a \sum_i a_i - \frac{1}{2} \sum_i \sum_j a_i a_j y_i y_j K(x_i, x_j) \quad s. t. \quad \sum_i a_i y_i = 0, 0 \leq a_i \leq C, \forall_i \quad \text{Equation 2-21}$$

The support vectors usually correspond to a small fraction of the training set and correspond in non-zero alpha coefficients a_i ($a_i \neq 0$) in the Lagrangian optimization problem. The support vectors form the SVM classification model since the normal vector w cannot be computed directly as shown in Equation 2-10. This is due to the fact that there is no vector in the n -dimensional space that maps directly to w which lives in the m -dimensional space. Hence, SVMs lead to a sparse classification model where only the support vectors need to be processed. Furthermore, the decision of which training samples are support vectors is automatically made and there is no need to explicitly determine the model such in the case of neural network [19]. For this reason SVMs have developed into a widely used pattern recognition tool and have been used in a wide range of diverse applications.

-
-
- Load y_i for each samples x_i and initialize a_i and b to a random value, & $C = \text{some value}$ (usually 10^n where n is a positive or negative integer)
 - Define: $E_i = f(x_i) - y_i$, where $f(x_i)$ is predicted output for input x_i
 - Repeat until *KKT* conditions are satisfied within a small margin epsilon ε for all a_i
 - i. Choose Lagrange multipliers to optimize
 - Find a Lagrange multiplier a_1 that violates the *KKT* conditions for the optimization problem.
 - Choose randomly, prefer unbound examples $0 < a_i < C$
 - Choose a second Lagrange multiplier a_2 to maximize $|E_1 - E_2|$, or choose randomly
 - ii. Find solution to the reduced optimization problem:
 - Subject to constrains $y_1 a_1 + y_2 a_2 = k$ and $0 \leq a_1, a_2 \leq C$, where k is a constant that depends on the previous values of a_1 and a_2
 - iii. Update SVM by computing new threshold b

Figure 2-13. The SMO algorithm

2.5.2 Support Vector Machine Classification

A. Monolithic Support Vector Machines

After solving the SVM optimization problem the values for the alpha coefficients a are obtained. The vector w , however, that describes the separating hyperplane cannot be obtained analytically from Equation 2-10 because the mapping into higher dimensional spaces is done through the kernel function. Hence the training data samples need to be used to classify new data. However, for the majority of samples the alpha coefficients are zero, with non-zero alphas corresponding only to support vectors. Hence, only the support vectors are used for classification purposes. The SVM classification decision function (CDF) that is used to predict the class for an unknown input data z is shown in Equation 2-22, where N_{SV} is the number of support vectors, a_i are the alpha coefficients, y_i are the class labels of the support vectors, s_i are the support vectors, z is the input vector, $K(z, s)$ is the chosen kernel function, and b is the

bias which be estimated by substituting data sample and their label into the classification equation and finding the mean value of the bias.

$$\text{Classification Decision Function} - \text{CDF}(z): \text{sign}\left(\sum_{i=1}^{N_{SV}} \alpha_i y_i K(z, s_i) + b\right) \quad \text{Equation 2-22}$$

B. Multiclass Support Vector Machines

The SVM classifier and concepts described thus far target binary classification problems. The concept of SVMs can also be extended to handle multiclass problems. However, formulating the SVM algorithm to handle multiclass problems increases the computational complexity. Hence, as an alternative, two other common approaches are used to extend binary classification algorithms, including SVMs, in order to solve multiclass problems. The first is the *one – against – all* approach in which one group of samples belonging to one class is chosen as the positive class while all the rest are combined to form the negative class. Under this setup it is necessary to train n classifiers for an n class problem. A voting scheme is used then to determine the dominant class. The second method is *one – against – one* approach where a classifier is constructed for each class pairing. It is necessary to train $\frac{n \times (n-1)}{2}$ classifiers for an n class problem. The final classification result can be obtained by accumulating the number of times each class has been predicted. The one with the majority of predictions is chosen as the resulting class. It is possible to alter the SVM formulation to take into consideration n classes instead of just 2, however, it is a computationally more difficult problem as it increases the training set and the accuracy is not that much better than the other two approaches.

2.5.3 Computational Challenges of Support Vector Machines

The SVM classification decision function (Equation 2-22) must be calculated for every input vector that needs to be classified. To do so the kernel must be evaluated, which requires all vector components to be processed with the corresponding support vector, its outcome multiplied with the alpha coefficients and support vector class label, and accumulated to the total sum before the bias can be processed (a process shown in Figure 2-14). The number of operations needed to compute the kernel depends on the number of support vectors (the

number of operations increases linearly with the number of support vectors), and the vector dimensionality. In general the classification phase is of the order $O(N_{SV} \times d_v)$, where N_{SV} is the number of SVs and d_v is the vector dimensionality.

Now specifically, processing each input vector takes time proportional to the kernel computation time which requires both vector and scalar operations. The vector operations are usually proportional to the vector dimensionality $O(d_v)$, while the scalar operations denoted as d_s depends on the kernel function. Hence, the actual number of operations needed to compute the outcome of a kernel is $(N_{SV} \times (d_v + d_s))$. In the linear kernel case, the value of d_s is equal to zero. However, for the other kernels which have additional operations the value of d_s depends on the actual implementation and resources on the processing platform. Usually, however, the vector operations necessary are more than the operations needed for the processing of the scalar value, and thus dominate the overall computation of the SVM feed-forward phase.

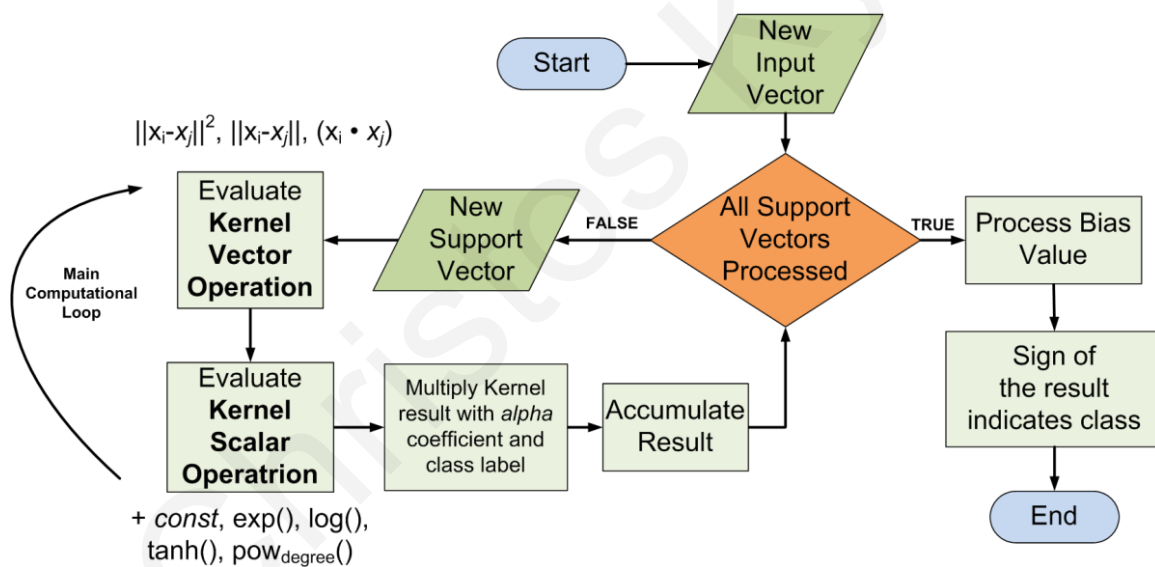


Figure 2-14. Support vector machine classification procedure

A. Improving the Classification Speed of Support Vector Machines

The classification time is proportional to the number of SVs since vector operations dominate the overall SVM computation flow. In turn the number of SVs is proportional to the size of the training set. In many real-world problems the training set can be quite large leading

to an increased size of SVs which results in slow classification speeds. Hence, one way to reduce the processing time is to reduce the number of SVs necessary to classify an input. Another approach is to reduce the dimensionality of each vector to obtain speedups [47], [48]. Literature suggests different ways to approximate the SVs with a fewer number of vectors, the most noticeable of which are outlined next.

One technique is to select a subset of SVs from the complete set and use only them to classify a new input [49], [50]. The key challenge for such methods is how to select the subset of SVs. The selection process aims at preserving the SVs which contribute the most to the SVM classification (Equation 2-22). The importance of each SV s_j can be measured by the mean square value [49], shown in Equation 2-23, over all the other SVs (the only samples with $a_i \neq 0$). Then the support vectors are sorted in decreased order and the first N_{red} SVs are selected to form the new subset, $N_{sub} < N_{SV}$ according to the ranking order. However, due to the change in the classification decision function vectors the old alpha coefficients a , and bias term b cannot be used since they are optimized for the complete SV set. Hence, the new parameters a_i^{sub} and b^{sub} need to be computed, resulting in a new classification equation show in Equation 2-24. However, in the case where all SVs have similar contributions leading to poor classification results even when removing only a few SVs.

$$\frac{1}{N_{SV}} \sum_{i=1}^{N_{SV}} [a_j y_j K(s_j, s_i)]^2 \rightarrow \frac{1}{N_{SV}} a_j^2 \sum_{i=1}^{N_{SV}} K(s_j, s_i)^2 \quad \text{Equation 2-23}$$

$$\text{CDF}_{sub}(z): \text{sign}\left(\sum_{i=1}^{N_{sub}} \alpha_i^{sub} y_i K(z, s_i) + b^{sub}\right) \quad \text{Equation 2-24}$$

Hence, there are other approaches which attempt to approximate all SVs with a different set. This approach was first proposed in [51] and improved in [52,53]. It is based on the idea that it is possible to approximate the decision surface using a *reduced set* of vectors (RSVs) s_i^{red} , $1 \leq i \leq N_{red}$, where $N_{red} < N_{SV}$. The reduced set method starts with a trained SVM and tries to find the reduced set of vectors through an approximated pre-image expansion. The SVs encode a vector w normal to the separating hyperplane in the feature space as shown in Figure 2-15. It is possible then to find a new set of vectors (of a much smaller size than the original set $N_{red} \ll N_{SV}$) that encode a new vector w^{red} that approximates the original vector

such that the distance p between the new and original vectors is minimum. A key point here is that although the vector w is not known explicitly and is computed through the kernel. Hence, the problem becomes to find the set of vectors that encode w^{red} .

-
-
- The normal vector w is given by the set of support vectors s_i , and weights a_i
 - i. $w = \sum_i^{N_{SV}} a_i y_i \varphi(s_i)$
 - To classify a new sample x one computes
 - i. $w \cdot x = \sum_i^{N_{SV}} a_i y_i K(x, s_i)$
 - Consider now the approximation vector vector w^{red} given by a set $s_i^{red}, i = 1, \dots, N_{red}$ and corresponding weights a_i^{red} , where $N_{red} < N_{SV}$
 - i. $w^{red} = \sum_i^{N_{red}} a_i^{red} \varphi(s_i^{red})$
 - Then it is desired that the approximation vector w^{red} is as close to w as possible and hence their difference p must be minimum
 - i. $p = \|w - w^{red}\|^2$
 - Hence one needs to find the vector set s_i^{red} and weights a_i^{red} in order to minimize
 - i.
$$\|w - w^{red}\|^2 = \sum_{i,j=1}^{N_{SV}} a_i a_j K(s_i, s_j) + \sum_{i,j=1}^{N_{red}} a_i^{red} a_j^{red} K(s_i^{red}, s_j^{red}) - 2 \sum_{i=1}^{N_{SV}} \sum_{j=1}^{N_{red}} a_i a_j^{red} K(s_i, s_j^{red})$$
 - Once they are found a new sample is classified using
 - $CDF(x) = \sum_i^{N_{red}} a_i^{red} y_i K(x, s_i^{red})$

Figure 2-15. Reduced-Set Method

This problem as formulated in Figure 2-15 is commonly solved using iterative, greedy, and gradient decent algorithms. Since the solution is not exact, there tends to be a small decrease in classification performance. However, it has been shown that in most cases it is not significant and these solutions work well for real-world problems. It must be noted that since the reduced-set-vectors (RSVs) are approximations the association with the original problem is lost, hence the support vectors no longer correspond to the original data interpretation (face image, digit image, etc). Finally, it has been shown in [51] that for the 2nd degree homogeneous polynomial kernel a solution can be found analytically for the desired number of reduced of support vectors. This makes it an extremely attractive kernel for embedded

applications since it offers a way to trade-off accuracy and processing performance based on the application demands. For other kernels such as the RBF an approximation of the reduced set may be found however, it is often infeasible and computationally very demanding.

B. Multistage Cascade Support Vector Machines

It is also possible to speed-up SVM-based classification by exploiting certain application characteristics such as that: (a) samples from one class may appear more frequently and (b) samples of one class may be more easily categorized than those of the other class. To this end, works in literature have tried to take advantage of these two observations by utilizing multistage SVM classifiers, with stages of classifiers of increasing complexity which are sequentially applied to the input data. Multistage SVM classification schemes include SVM classifiers that mostly follow a cascade structure [54], [55], [56], and are constructed and trained in the manner presented in Section 2.1.2. Another form of a cascade SVM is to not use different classifiers at each stage but a single SVM and assign a number of SVs at each stage. Thus the SVs of a single SVM are sequentially applied to the input data given that they pass a threshold at each stage [57], [58]. Other examples of multistage SVMs include tree structures [59], and ensembles of SVMs which utilize voting schemes [60]. These works tried to improve the classification times over single SVM classifiers demonstrating speedups as well as improved accuracy. The characteristic of such arrangements is that the early stages process the majority of the data and usually have low complexity, meaning that they require less SVs to be processed, and as such take less time to process. Conversely, the latter stages have higher complexity as they require more SVs to be processed, and hence have a longer classification-time. The SVM speedup methods outlined in the previous section can be used in combination with cascade SVMs to reduce the processing time of each individual SVM stage.

2.6 Complimentary Material

This section outlines complementary material that is relevant to the research presented in this thesis.

2.6.1 Feature Extraction

To recognize or classify an object in an image, one first needs to extract some features out of the image, and then use these features inside a pattern recognition algorithm to obtain the final classification results. Feature extraction (or detection) aims to locate significant visual feature in image regions depending on their intrinsic characteristics and applications. These features can be defined in global or local neighborhood and distinguished by shapes, textures, sizes, intensities, statistical properties, and so on. Feature extraction algorithms are considered as a necessary step, in order to deal with issues stemming from object variations due to illuminations and cluttered environments. The goal of feature extraction algorithms is to choose those good features that allow data belonging to different categories to occupy compact and disjoint regions this allowing machine learning algorithms to form improved decision boundaries. Hence, there has been increasing research interest in research on discriminative features that can lead to more efficient detection.

The Haar-like features, described in detail in Section 2.4.2, are very attractive because they are simple features that can be computed at any position and scale in constant time, given that the integral image is computed, and thus allow for fast rapid processing. However, a large number of Haar-like features are necessary to reach the desired detection/false acceptance rate trade-off. It results in a long training procedure [54] and cascades with several stages which are difficult to design. Furthermore, Haar-like features are not robust to local illumination changes. Alternatively, two other popular approaches for visual object detection which have been widely adopted in recent years include Histogram-of-Oriented Gradients (HoG) [61] and Local Binary Patterns (LBP) [62]. In both cases histograms are constructed from features extracted from smaller image blocks. However, when considering embedded vision systems LBPs are a very attractive option because of their low computational complexity while providing the necessary invariance to local illumination changes compared to Haar-features.

Other approaches that are described include edge detection features and histogram equalization.

A. Local Binary Patterns (LBP)

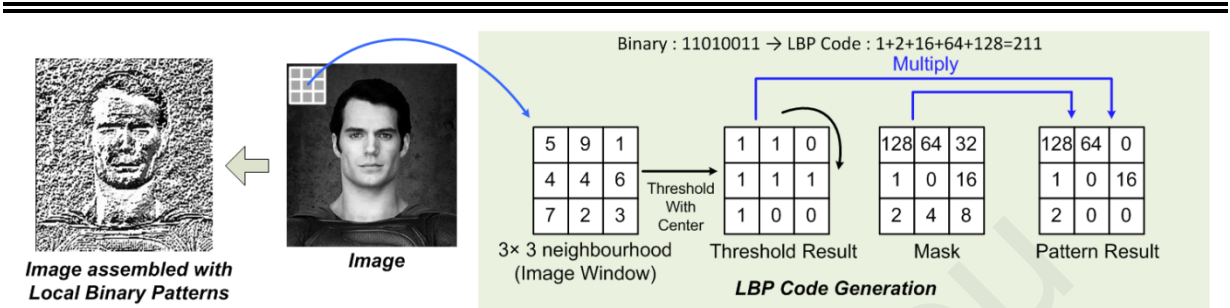


Figure 2-16. Local binary pattern code generation process

Local Binary Patterns (LBPs) have been used for a wide range of applications ranging from face detection [63], [64], face recognition [65], facial expression recognition [66], pedestrian detection [67], to remote sensing and texture classification [68] amongst others in order to build powerful visual object detection systems [69]. LBPs are local patterns that describe the relationship between a pixel and its neighborhood. Many variants of LBPs have been proposed in literature[70]. The most common approach however, dictates that each 3×3 window in the image is processed to extract an LBP code. The processing involves thresholding the center pixel of that window with its surrounding pixels using the window mean, window median or the actual center pixel, as thresholds. Then the LBP code is given by Equation 2-25, where I_{thresh} is the chosen threshold value and I_n are the intensities of the surrounding window pixels with ($n = 0,1, \dots,7$).

$$LBP \text{ code} = \sum_{n=0}^7 \text{step_fun}(I_n - I_{thresh}) \times 2^n, \quad \text{step_fun}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad \text{Equation 2-25}$$

The overall process consists of the following steps (shown in Figure 2-16): **1)** Threshold the values in a 3 × 3 neighborhood (image window) with the chosen threshold placing 1 where the value is greater or equal than the threshold and 0 otherwise. **2)** Multiplying the resulting binary map with a predefined mask (usually incremental powers of two). **3)** Sum the values to obtain an 8-bit LBP Code.

The result of LBP processing is an image assembled by LBP features. The next step to creating an LBP-based descriptor requires dividing the LBP-based image in k_{LBP} blocks of $W_{width}^{LBP} \times W_{height}^{LBP}$ pixels (e.g. $2 \times 4, 4 \times 4, 8 \times 8$). A local histogram is generated for each block in the image in order to build local image descriptors. The local histograms are then concatenated to form a single global histogram, as show in Figure 2-17. The global histogram approach effectively expresses information in three different levels: the individual LBP code contains information at the pixel-level, the local histograms contain information on a regional level, and the concatenated regional histograms contain a global description. As such, the resulting histogram encodes both local and global characteristics in a compact representation which makes it more robust to object pose and illumination variations.

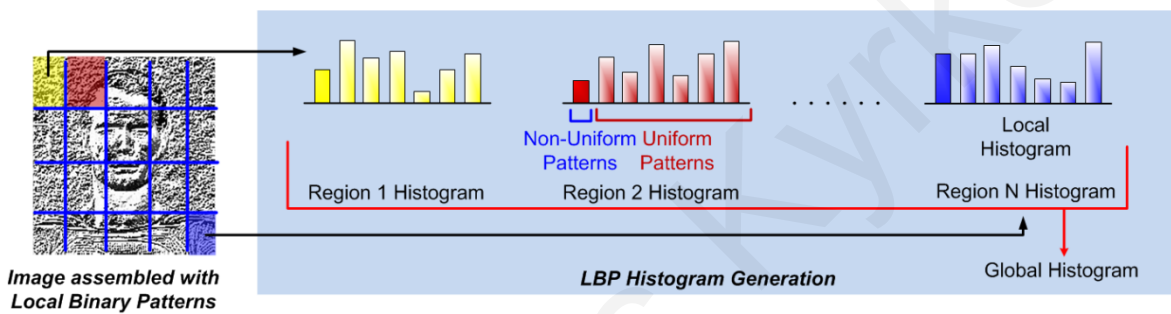


Figure 2-17. Local binary pattern histogram generation process

Each local histogram measures the occurrence of each of the 256 possible LBP codes in the block. The number of bins necessary for the histogram description can be determined by the resulting accuracy after the training and testing phases. When all 256 bins are used this will result in a long histogram descriptor which will have both high computational and storage demands. Alternatively, Ojala et al, propose the concept of uniform and non-uniform patterns to both reduce the number of possible LBP patterns and improve discrimination power. An LBP pattern is called uniform if it has 2 or less transitions e.g. 11110000, and non-uniform if it has have more than 2 transitions e.g. 10100101. It was observed that textured images are consisted mostly by uniform patterns. This also applies for object images since they can be seen as the composition of micro-textures. Hence, the histogram is divided into 59 bins instead of the possible 256, where one bin is devoted for all non-uniform patterns, and the rest are assigned to the remaining 58 uniform patterns. The general histogram descriptor can be used

to feed an adequate classifier or discriminative scheme, such as support vector machines, in order to perform object detection.

B. Edge Features

Edge detection refers to image processing operations which aim at finding image regions with sharp illumination discontinuities [71]. Edge detection is a fundamental tool in image processing, machine vision and computer vision, particularly in the areas of feature detection and feature extraction. Edges are important features in images that can convey information about structures of objects and surfaces within a scene such as boundaries of objects or overlapping items [71]. As such edge detection methods are commonly used in many image processing and computer vision applications including visual object detection [72],[73]. in order to significantly reduce the amount of processed data and filter out unnecessary information. Their major characteristic and is that they typically result in binary images and thus can be processed very efficiently and also reduce storage requirements. Edges are identified by finding regions of sharp illumination changes within an image, however, in itself not a trivial task. In general the edge detection methodology uses an edge operator which can rely on mask approximations or first and second order image derivatives applied to both the spatial or frequency domains to find areas of sharp transitions [74]. Then these areas are compared against a threshold to remove false or weak edges. This process is outline shown in Figure 2-18 which also shows common edge operators. The threshold affects the number of edges that will appear in the edge image and the susceptibility of subtle changes and noise and a suitable threshold is determined experimentally.

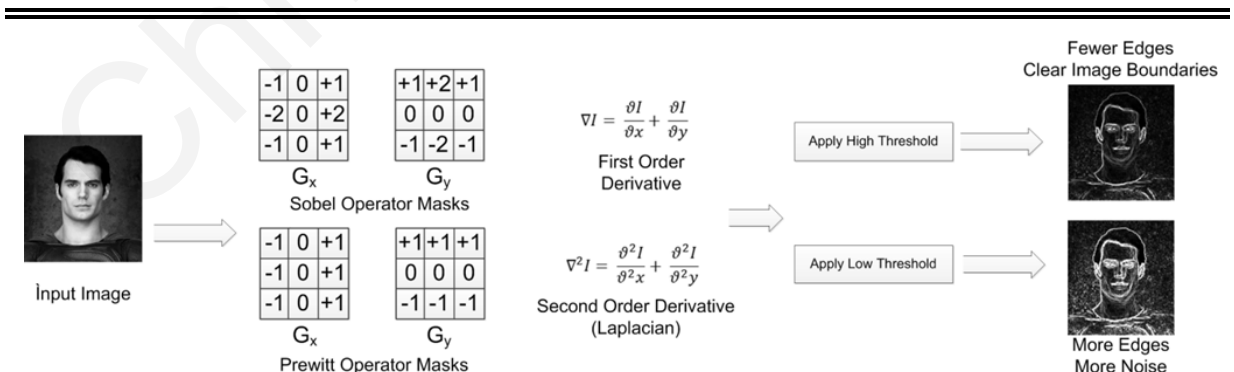


Figure 2-18. Edge Detection Process

C. Histogram of Oriented Gradients

Histograms of Oriented Gradients (HoGs) [61],[75] are feature descriptors which attempt to capture object appearance and shape by finding how the different gradient orientations are distributed in an image, and are thus able to capture the shape of an image. Hence, they have successfully been used for non-rigid objects with flexible silhouette such as pedestrians [61], and are often coupled with a Support Vector Machine (SVM) classifier.

The process to generate the HoG descriptor, show in Figure 2-19, begins by first computing the 1st order horizontal and vertical gradients $f(x, y)$ of an image $i(x, y)$, using Equation 2-26. This is necessary in order to find the magnitude $m(x, y)$ of the gradients as well as their direction $\theta(x, y)$ for each pixel location (x, y) using Equation 2-27 and Equation 2-28 respectively. After the computation of this two values the image is separated into cells where each location has a respective gradient magnitude and direction, and for each cell a histogram is computed. The histogram is generated as follows: 1) select the bin where each location belongs to using the direction $\theta(x, y)$. 2) increment the value of that bin using the magnitude $m(x, y)$. 3) Group the cells into rectangular or circular bocks and normalize each histogram cell using Equation 2-29, where V_k is a vector corresponding to the combined histograms within the block and v is the normalized histogram. 4) Concatenate the normalized histograms v of each block to form the histogram descriptor. The parameters specific to the HoG descriptor such as the cell size, block size, number of histogram bins and overlap between cells and blocks are determined experimentally and are different for different object detection applications.

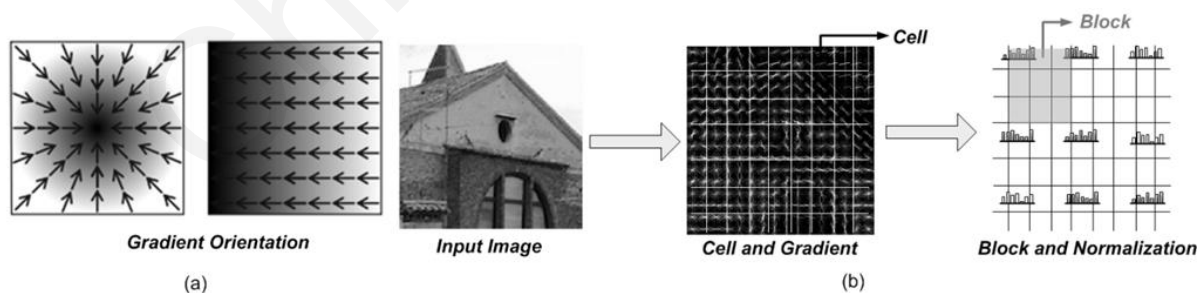


Figure 2-19. Histogram of Oriented Gradient

(a) Example of orientations in a gradient image. (b) The HoG descriptor process.

$$\text{Gradients} \begin{cases} f_x(x, y) = i(x + 1, y) - i(x - 1, y) \\ f_y(x, y) = i(x, y + 1) - i(x, y - 1) \end{cases}, \text{ where } i \text{ is the image} \quad \text{Equation 2-26}$$

$$m(x, y) = \sqrt{f_x(x, y)^2 + f_y(x, y)^2} \quad \text{Equation 2-27}$$

$$\theta(x, y) = \arctan\left(\frac{f_x(x, y)}{f_y(x, y)}\right) \quad \text{Equation 2-28}$$

$$v = \frac{V_k}{\sqrt{\|V_k\|^2 + 1}} \quad \text{Equation 2-29}$$

D. Histogram Equalization

Histogram equalization is a non-linear gray-level transform function which is applied on an input image in order to yield a new image of enhanced quality. In the object detection process it is performed for every search window prior to the classification process in order to compensate for differences in illumination, brightness, different cameras, and contrast variations [76]. Thus reducing within class variability by enhancing common features in images (Figure 2-20) thus leading to more accurate classification results.

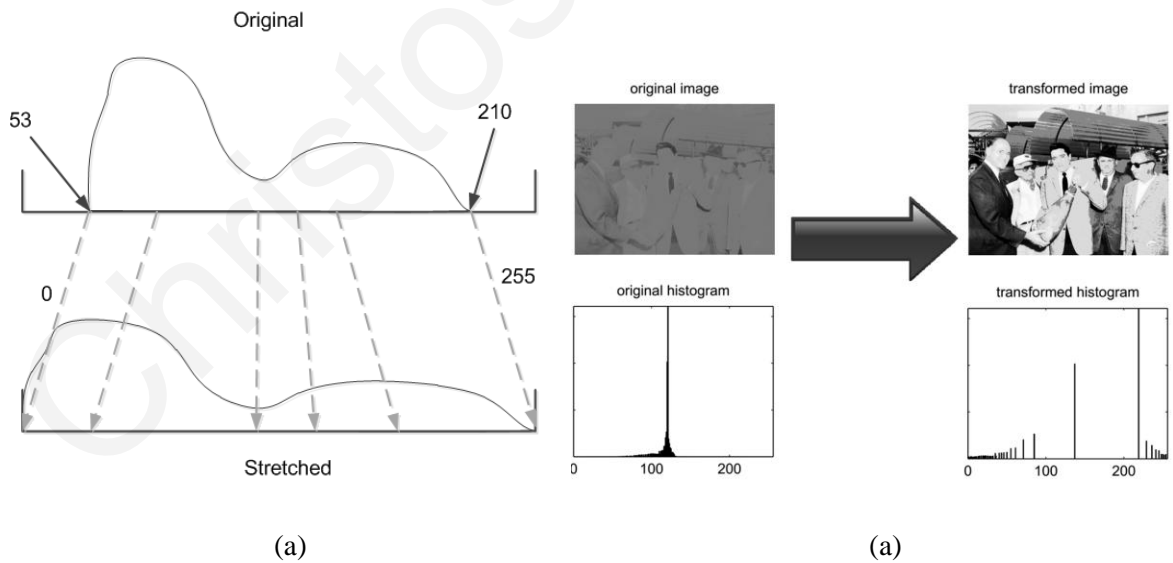


Figure 2-20. Histogram Equalization

(a) Histogram Stretching (b) Example of the effect of histogram equalization

The goal of this process is to "stretch" the histogram (Figure 2-20) of the image so that the pixel values that appear in the image appear throughout the entire spectrum of possible values. So the goal of the histogram equalization algorithm is to find a gray-level transformation T which makes the histogram flat. The histogram of an image x is a discrete function $h(r_k) = n_k$ where r_k is the k^{th} pixel gray level, and n_k is the number of pixels in the image having the gray level r_k . Hence, An estimate of the probability p of occurrence of gray level r_k can be found by normalizing the histogram thus deriving $p(r_k) = n_k/n$, where n is the total number of pixels in the image ($n = IM_{width} \times IM_{height}$). In order to obtain the transformation function we first need to find the cumulative distribution function (*cdf*) which for every pixel value r_k is given by $s_k = T(r_k) = \sum_{j=0}^k P(r_j)$ which essentially amounts to summing the estimated probabilities of occurrence up to value k . This will map all pixel values r_k to new values s_k which will have a stretched and almost flat histogram. However, the s_k correspond to cumulative probabilities and as such are within the range (0 – 1) so a final transformation needs to take place to remap the values to (0-255) form the new image x' giving leading to the transformation in Equation 2-30. The results is rounded to produce an integer pixel value.

$$x'(i,j) = round(T(x(i,j)) \times [(max(x) - min(x)) + min(x)]) \quad \text{Equation 2-30}$$

2.6.2 3D Stereo Computer Vision

Stereo vision is a technology for extracting the depth of objects in a scene by matching the left and right images taken from a stereo camera setup. Therefore, it is commonly employed in emerging embedded applications such as surveillance, autonomous vehicles and mobile robots in order to increase contextual awareness of computers and vision machines.

Depth information can be extracted by using a stereo vision system which works similarly to the way that a biological visual system infers depth. Stereo vision systems can infer depth information about a scene from a stereo image pair (usually referred to as left and right images) [77]. Depth information evolves from the computation of the disparity map, from which information about the depth of objects in an image frame, relative to the location of the

camera(s), can be extracted. There are three important tasks for computing the disparity map: *camera calibration*, *stereo image rectification* and *stereo correspondence*.

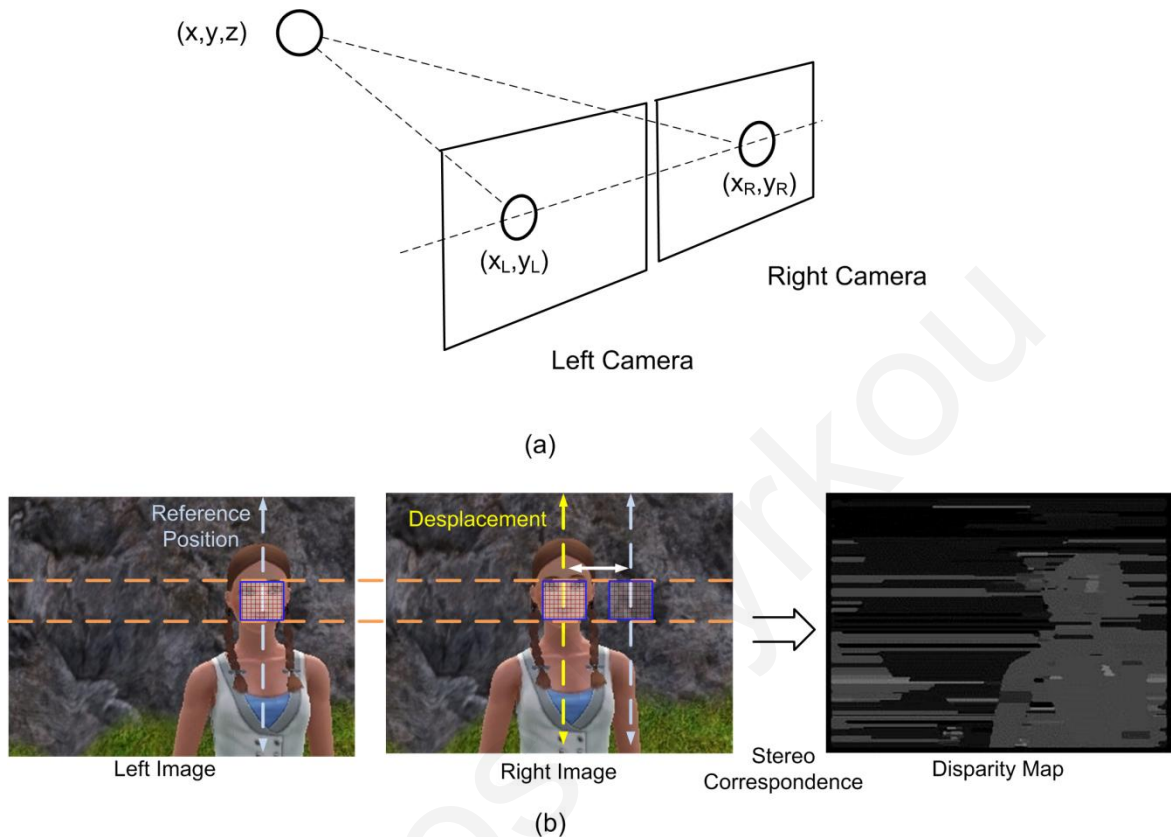


Figure 2-21. Stereo setup and depth computation

(a) Stereo setup with each camera capturing the same object (scene) but from a different angle. (b) Computation of depth from a stereo image pair by measuring the displacement from one image to the other.

Calibration integrates information from intrinsic camera parameters such as focal length (f), and extrinsic parameters such as relative orientation angles between the two cameras, to determine the camera perspective projection matrices, necessary for the stereo rectification algorithm. Rectification can project one of the two images in the other's common plane to reduce a $2D$ search problem in non-rectified images into a $1D$ search problem along the horizontal raster lines of the rectified images [77]. The stereo correspondence algorithm takes the two rectified images as input and produces a disparity map for each pixel in one of the two images (Figure 2-21). Generally, the disparity map stores distances between corresponding points of interest in the two images, and is computed using searching and matching

techniques. First, it is necessary to locate a common point of reference in the two images. This is done by a small window which horizontally scans both images and compares the left and right regions. The location which exhibits the highest similarity between the left and right images is then used to derive the disparity map. Depth information can then be computed from the disparity map by triangulation using the focal length and the baseline distance between the stereo camera optical centers. The extraction of depth using a stereo camera system can make it possible to combine depth information with traditional 2D object image processing and computer vision techniques in order to increase the awareness the capabilities of a visual processing system.

CHAPTER 3

A FLEXIBLE PARALLEL HARDWARE OBJECT DETECTION ACCELERATOR FOR THE ADABOOST-BASED HAAR-FEATURE CASCADE

The AdaBoost-based object detection framework, presented in the seminal work by Paul Viola and Michael Jones in 2001 [15], is widely considered as a state of the art in terms of object detection and has been integrated into many object detection systems through its implementation in the Intel® Open Source Computer Vision library (Open CV) [18]. The framework incorporates the integral image representation, cascade classification scheme, and simple rectangular features called Haar-features in order to improve processing speed. The capability for real-time processing is a very important for a wide range of embedded applications such as automotive, surveillance, and interactive entertainment. Though a number of techniques are used in the framework to speedup processing, real-time performance on resource-constraint, low-power embedded systems is still difficult to achieve because of the computationally intensive nature of the algorithm. Hence, some type of acceleration is necessary to take advantage of the huge amount of parallel computations so that it is possible to achieve real-time performance while meeting additional constraints.

3.1 Related Work on Haar-Feature AdaBoost-based Classifier Cascade Implementations

Recent attempts to accelerate the Viola and Jones proposed visual object detection process can be summarized with respect the acceleration approaches. The first, is to develop parallel software that attempts to take advantage of the multiple processing cores found on modern general-purpose computing platforms with such as CPUs and GPUs. Implementations of the AdaBoost algorithm on multi-core CPU systems [78]¹, [79]² for pedestrian and face detection respectively, maintain a high degree of flexibility, however, they do not fully exploit the

¹ Two Intel® Core™ 2 quad processors @ 2.33 GHz

² Intel Core i7-2655LE CPU @ 2.2 GHz and 4GB RAM

available parallelism (i.e. speedup is not proportional to the number of cores) due to load imbalance and synchronization overheads which limit the utilization of the cores. Thus CPU implementations rely on limiting detection to small image sizes and object sizes [78] (using a machine with), or permit only a fixed number of objects to be present in the image [79] (run on an) to provide real-time processing. In addition, multi-core processing systems have high power consumption demands. Alternatively GPU platforms have a much richer pool of parallel computing resources to rely and are designed to execute independent parallel tasks (kernels). Consequently, related GPU implementations [27]¹, [80]² both targeting face detection, manage to achieve real-time processing frame-rate when maintaining high core utilization and make efficient use of the GPU memory hierarchy. However, GPUs need to be controlled by a host CPU which increases communication overheads and data transfers between CPU and GPU memories, and also operate at high power levels which restricts their use in embedded domains. Furthermore, feature upscaling is not efficient since it underutilizes the cores which decreases performance,

The second approach, is ASIC implementation which is on the opposite end of the spectrum in terms of flexibility compared with general purpose platforms, but is tailored made for the targeted application thus offering the highest performance with the least power consumption and gate count. An example is the implementation in [81] where a specialized recognition processor was presented. However, ASIC design is costly and the fixed circuitry reduces its flexibility which is necessary for many embedded applications.

Third, is the implementation of the algorithm on FPGA platforms [82], [83], [84], [85], [86], [87]. This approach is attractive for embedded applications since it offers lower power consumption compared to GPUs with the parallel processing capabilities of a dedicated ASIC architecture but with increased flexibility through reconfiguration. The majority of these FPGA implementations are based on array processing architectures which offer both parallel data access and processing. One of the first works to show the benefits of array processing architectures is the work in [82], and suggests the computation of the integral image as well as

¹ NVIDIA GeForce GTX 285 GPU (30 cores, 16 KB shared memory per core, and 1GB RAM)

² NVIDIA GTX 470

the access of its values to be implemented as a systolic array rather than using a central, or even distributed, memory. The work by M. Hiromoto et al [83] proposes a hybrid model consisting of parallel and sequential modules and the goal is to reduce the necessary computational resources by assigning more resources to the parallel modules and less to the sequential ones. The parallel modules are assigned to the early stages of the algorithm which are frequently used whereas the latter stages are mapped onto sequential modules as they are executed less frequently. However, the split point between the stages executed on the parallel and sequential modules becomes a critical design choice and needs to be reevaluated for a different cascade which may lead to performance/accuracy degradation. Furthermore, the parallel modules may still need considerable amount of resources even with this scheme. The work in [84] utilizes an architecture that is based on register different array structures that perform integral image computation and classification per single window. A window is fed to the array-based integral image computation module through a scan-line buffer which holds part of the image. The integral image pixels are processed in parallel by the Haar-feature classifiers. Up to three classifiers can be instantiated in parallel in order to boost performance. In the work presented by Y. Shi et al [86] proposes an array-based architecture for a single window that introduces two pipelines to increase the detection process. The vertical pipeline in the array computes the integral image and the horizontal pipeline computes a rectangle feature in one cycle. However, the architecture was only evaluated on small scale images. The implementation in [87] employs very similar architecture to the one presented previous works but with a massively reduced number of training features which allows the architecture to exceed 100 FPS, illustrating that training set optimizations are also a factor that can be potentially explored. However, a major drawback of that work was that it provided no information on the method used to reduce the training set or its impact on the overall detection accuracy. A simpler version of the AdaBoost-based Haar-feature Cascade was implemented in [85] where only 3 cascade stages are used, with an input image size of a 120×120 image. The integral image is computed for each sub window that is generated, rather than once for the whole image. Furthermore, an image pyramid generation unit is used to produce downscaled versions of the input image. The implementation in [7] demonstrates a complete vision system based on the AdaBoost Haar-feature cascade that also incorporates skin and motion detection to reduce the search space. The integral image is generated for the whole image and a single

window is extracted from the memory and processed in parallel for all window sizes. However, it downscales the input 640×480 image to 80×60 image to achieve real-time performance while considering only three object sizes. Table 3-1 presents a summary of existing implementations, and gives a brief comparison in terms of the training sets used (features and stages) and methodologies.

TABLE 3-1: ALGORITHM AND METHOD COMPARISONS FOR RELATED FPGA WORKS

Related Works		Hiomoto [83] ^a	Cho [84]	Wei [85]	Shi [86]	Lai [87]	He [7] ^b	Presented Architecture
Cascade information	Features	2,913 [18]	2,135	225	2,913 [18]	52	115	2,913[18]
	Stages	25 [18]	22	3	25 [18]	1	9	25 [18]
	Feature Size	24×24	20×20	24×24	24×24	20×20	11×11, 19×19, 27×27	24×24
Image Size		640×480	320×240, 640×480	120×120	176×144	640×480	80×60	320×240
Scaling Method		Image Downscaling	Image Downscaling	Image Downscaling	N/P	Image Downscaling	Image Downscaling	Image Downscaling and Feature Upscaling
Downscale Factor(s)		1.2	1.2	1.25	N/P	1.25	None	1.25, 1.33, 2
Image Scales		18	14 (320×240), 18 (640×480)	4	N/P	~15	3	3 downscaled images, 5 upscaled features
Integral Image Computation		Per window, using a register-array	Per window, using a register-array	Sequential computation per window	Per window, using a register-array	Per window, using a register-array	For the whole image	For an image region
Rectangle & Feature Computation		First 10 stage in parallel, other sequentially	Up to three feature classifiers in parallel	Single feature at a time	Single feature at a time	Single feature at a time	A single feature of all window sizes.	A single feature for multiple windows

^a Uses a sequential and parallel processing execution model. Split point is at stage 10.

^b Includes skin and motion detection to reduce search space

N/P - Not Provided

Overall, the majority of FPGA related works use a fixed feature size to process the whole image and its downscaled versions. Furthermore, they rely on the rapid processing of a single window in order to achieve real-time performance. They utilize register array buffers to compute the integral image and Haar-features in parallel. Under this approach in order to

maintain the frame-rate as the number of windows increases either the frequency needs to be increased or more hardware will be necessary to increase parallelism. In both cases power there will be an increase in power consumption and required hardware resources and as a result this approach will not suffice for embedded applications. Alternatively, the other approach is to design an architecture that will be optimized to process many windows in parallel, with less parallelism afforded to the Haar-feature computations. However, there are challenges in designing such an architecture which are outlined in the following section.

3.2 Mapping Algorithm to Hardware

3.2.1 Opportunities for parallelism

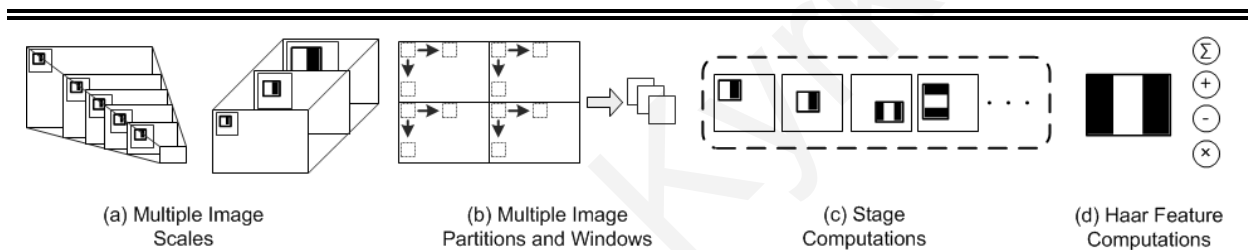


Figure 3-1. Parallelization in the AdaBoost-based object detection

Available parallelism in the AdaBoost detection Framework: **Coarse-Gran Parallelism:** (a) Different image scales can be explored either by upscaling the feature and rescanning the image or by downscaling the image and scan each scale with a single feature size. (b) Each scale factor generates multiple windows. Also each image can be partitioned to be processed independently. **Fine-Gran Parallelism:** (c) The features within a stage can be computed in parallel. (d) The processing required by features and rectangles can be executed in parallel.

An analysis of the algorithm, which has also been discussed in Section 2.4.4, can provide an insight on the available parallelism thus outlining what the characteristics and requirements of the hardware architecture need to be. The parallelism can be classified in the two groups, coarse-grained and fine-grained, illustrated in Figure 3-1. The coarse-grained parallelization includes processing different scale factors, different image partitions and multiple windows. On the other, hand, the parallelization of Haar-features in the same stage as well as feature and rectangle computations belong to the fine-grained parallelization. While the latter form has been the focus of most related FPGA works in the literature, FPGAs provide the means to also

take advantage of coarse-grained parallelism which will be necessary as the amount of windows increases (i.e. image size increases).

3.2.2 Hardware Architecture Requirements and Mapping

In order to design an architecture that can take advantage of the different forms of parallelism that are available in the AdaBoost-based Haar cascade detection algorithm, first it is necessary to consider the impact of each parallelization approach on the amount of computing and memory resources needed, as well as how flexibility is affected.

A. Integral Image Computation

The input image first needs to be transformed as an integral image so that the Haar-features can be computed more efficiently. One challenge in the implementation of the integral image computation however, lies in the implementation of the addition and storage required for computing and storing the integral image values. As the size of the input image grows, the adder and storage grow proportionally as well. Recall that an integral image pixel located at (x, y) holds the sums of all pixels above y and to the left of x (Figure 2-8). As the range of x and y grows, the amount of pixels summed for computing the value of integral image pixel (x, y) grows exponentially ($x \times y$), hence, the adder precision and memory requirements change as well. This can be addressed by applying the algorithm over smaller regions of the input image rather than the entire image. Given that the bulk of the computation focuses on computing each feature, and given that the computation is identical, the challenge when processing multiple windows shifts to finding an efficient way to access the values of the integral image in parallel, and be able to employ the inherent parallelism of computing these rectangles over the entire image. Thus, storing the integral and integral squared images into single memory blocks essentially limits the number of rectangle coordinates accessed in parallel and creates contention. Similarly, replicating the memory blocks to increase parallelism results in high memory requirements, and if the memory is off-chip, in an increased latency. Overall, since the inter data computations of the image preprocessing and integral image generations are regular and independent, they are fit to be accelerated through a

systolic-array inspired computing architecture, which allows parallel data movement and calculation the pixels in an image block.

B. Haar-Feature Evaluation

Since the feature value calculation of each Haar-like feature is independent and large amounts of Haar-like features will be generated based on the sub-windows, the computations of the feature value can be processed in parallel. This however, increases the need for computational resources as well as on-chip storage requirements and parallel access of the Haar-feature data. The computations of the Haar-feature need to also take into consideration the window size in order to support the enlargement of the searching window. Since the sizes of the windows are different as the feature windows are scaled up, it requires the array to have flexible scaling capability to also support the scaling of the feature windows along with image downscaling. These processing tasks include different operations, and hence require the array to have the flexibility, in terms of arithmetic and logic units, to change the computing logics to meet the requirements for processing an image region with different training set setups (different number of stages, number of features and number and position of rectangles per feature).

C. Illumination Compensation

The integral image representation benefits both the feature computation and the illumination compensation technique outlined in Section 2.4.3, that adapts the original feature threshold t_o to a compensated image threshold t (Equation 3-1) which dynamically takes care of any lighting variations encountered during the detection stage. It is needed to be done only once for every search window however, as shown in Equation 3-2, it requires to compute the standard deviation of the window σ which in turn requires that first the window variance VAR is found using the sum of an area of pixels.

$$t = t_o * \sigma_i \quad \text{Equation 3-1}$$

$$VAR = \sum_{i=0}^{\# \text{ of pixels}} \left[\frac{\sum_{i=0}^{\# \text{ of pixels}} X_i}{AREA} \right]^2 \quad \text{and } \sigma = \sqrt{VAR} \quad \text{Equation 3-2}$$

$$t = t_\sigma * \sigma_i \Rightarrow t^2 = (t_o)^2 * VAR \quad \text{Equation 3-3}$$

Division and the square root, which are the necessary operations for these calculations, are tedious tasks when implemented in hardware, hence, better solutions need to be explored. Given that the search window size is known, the costly division operation can be avoided using the reciprocal of the area as a constant, and multiply it. The sum of the pixels is then squared, and subtracted from the computed value of the squared integral image, to give us the variance. To compute the standard deviation, the square root of the variance is needed and to compute the *compensated threshold* the product of the original threshold and the *standard deviation* (σ_i) are needed. Therefore both sides of the equation can be squared, multiplying the variance with the squared value of the original threshold (can be pre-computed during training and stored in the training set), to yield the squared value of the compensated threshold. Thus, the square root operation becomes a multiplication instead.

The above considerations are addressed through the proposed hardware architecture for AdaBoost-based Haar-feature cascade object detection detailed in the following section, which aims to provide massive parallelism by simultaneously processing multiple windows both for integral image as well as Haar-feature computations. It also provides a hybrid scaling mechanism utilizing both image downscaling and feature upscaling.

3.3 Proposed Hardware Architecture

The hardware accelerator for the Viola-Jones detection framework consists of two major blocks, an image pyramid generation (IPG) unit and a systolic array which implement the different tasks in the algorithm. The IPG receives the input video frame and generates image regions which are examined for the targeted object(s) of interest by the systolic array processor. Each search window in the extracted image region is then processed in parallel, feature by feature and stage by stage through the systolic array. The array is responsible for computing the integral image, computing the rectangles for each feature, and evaluating both the features and the stage sums. If an object of interest is detected within that image region, the array outputs the coordinates of the window within the region which contains the object, taking into consideration the scale of the feature that the object was found at. When the array completes the examination of an image region, a new image region is fed into the array from the IPG unit, until the entire image is searched. The original image is also downscaled through

the IPG unit, producing more image regions for each downscaled version of the original image, until the downscaled image equals the search window size. This creates a hybrid architecture that evaluates features over a number of image scales, and a number of feature sizes over a single search window. A brief description of each of the units is given next, while a block diagram of the system architecture is shown in Figure 3-2.

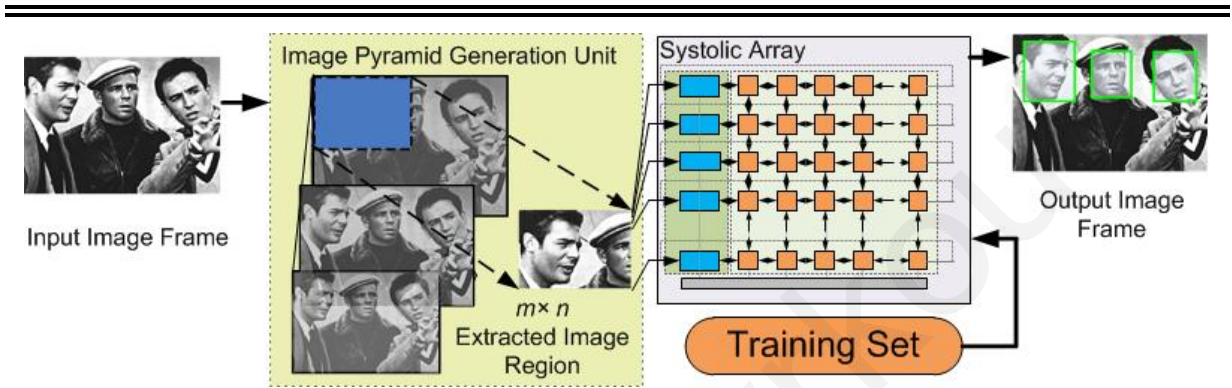


Figure 3-2. Systolic-array inspired accelerator architecture with the two major units

3.3.1 Image Pyramid Generation (IPG) Unit

The IPG unit, shown in Figure 3-3, receives the input video frame and extracts image regions to be processed by the systolic array. The unit receives pixels row-wise, and generates $m \times n$ image regions, which are then buffered and fed row-wise in parallel in the systolic array. The size of the extracted image regions is determined by the size of the systolic array. The IPG and the systolic array operate in a pipelined fashion, where the systolic computation happens as soon as an image region has been generated. However, the IPG continues to generate search window pixel data while the systolic array is computing, preparing the next search window(s) that will be used. Typically (depending on the systolic array size and subsequently search window size), the IPG can generate a second search window before the first one is computed by the array, therefore one search window buffer is sufficient.

The IPG unit also downscales the original image, ensuring that objects bigger than the search window size are downscaled, and eventually can fit into a search window as well. In this way, the search window can be made as large as the silicon budget allows the systolic array to be. Moreover, data loss due to downscaling is limited, as the image will not be scaled

down after it reaches a certain size. Instead a hybrid scaling mechanism is incorporated that utilizes traditional input image downscaling, as well as the original feature upscaling scheme that Viola and Jones proposed. This feature upscaling continues to iterate, in order to detect objects in the search window larger than the feature size. Iterations continue until the feature size equals the size of the smaller dimension of the search window size (typical features are square, whereas search window sizes can be rectangular). For example, when considering a starting feature size of 24×24 , and a search window size of 80×60 , features will be scaled up until the feature size will be 60×60 . In this way, there is a reduction in the image size where features are being evaluated, reducing the overall cost and amount of computation, and still allow the system to process large input images.

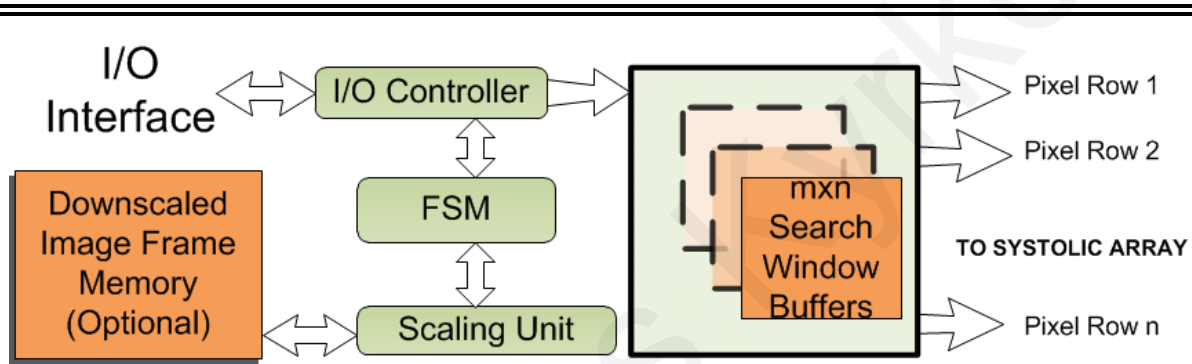


Figure 3-3. Image pyramid generation unit

The IPG unit consists of three stages, the input stage, where pixels are received from the frame memory, the partitioning stage where incoming pixels are partitioned into the search window buffer, and the scaling stage. The first stage is customized to satisfy the input conditions (i.e. number of pixels per cycle, etc.). The second stage is a finite state machine that is responsible for generating the address of the pixel values that are to be received in the next I/O operation and directs incoming pixels in their corresponding search window buffer location. Lastly, the scaling stage simply computes the coordinates (and subsequently memory address) of the downsampled image for each incoming pixel, generating the address where each pixel is to be stored. It must be noted that depending on the choice of the downscaling algorithm used, some pixels will be mapped to the same location. In the proposed IPG, the algorithm used is a simple multiplication, and the pixel that was lastly computed to be stored in the generated location, overwrote any previously written pixels. Additionally, the

downscaled image (depending on the generated size and thus its memory requirements) can be stored either on-chip or on external memory, and retrieved at a later stage during the computation in similar fashion as the original image is received. This procedure was chosen to enable flexible scaling of downsized images, allowing the designer to select the scale and the number of produced downsized images. The output search windows are fed pixel by pixel, row-wise, in the systolic array.

3.3.2 Integral Image and Haar-Feature Processing Array

The systolic array performs the bulk of the computation, it computes the integral image, collects and computes the rectangle points, computes and evaluates the feature and stage sums, and determines whether a region passes a cascade stage so that it can be considered for further search. The array also maintains the location of detected objects. The array consists of two types of processing elements (PEs), the collection and computation units (CCUs), and the evaluation units (EUs). The EUs are placed as the leftmost PEs in each row in the array, the CCUs make up the rest of the array. Each EU communicates via a direct link to its neighboring CCUs, and a toroidal link to the far right CCUs, as shown in Figure 3-4. The array also contains distributed control units (CUs), small FSMs that direct the overall operation by global control signals. The distributed control units also maintain the temporal consistency of the entire operation, acting as coordinators throughout the entire computation. Given the modular operation of the array, and the identical operation of each of the CCUs and EUs respectively, the control units maintain that communication is uniform across the array and towards the necessary direction, and that all units are synchronized, either doing data transfer, or computation. Distributed multiple CUs can be used, in order to reduce the size of the control region for each CU, since the control signals delivered to the PEs are identical. The array units can communicate with each of its neighbors via bidirectional data links.

The chosen array size depends largely on the application requirements in terms of frame rate and budget, and the training set and feature sizes used in the detection algorithm. The minimum size has to match the original size of the features, whereas the maximum size of the array can be determined based on the silicon budget available. The dimensions of the array can be made to match the proportions of the extracted image region to be processed.

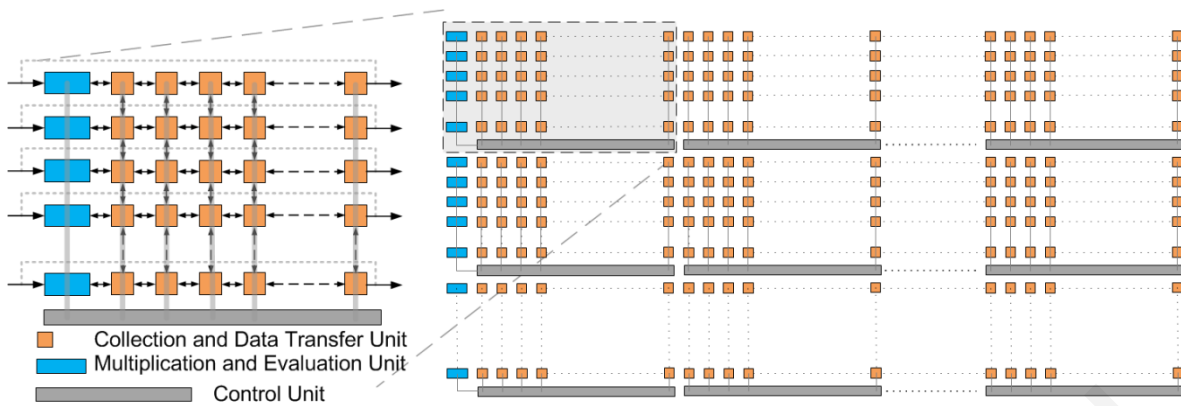


Figure 3-4. Systolic processing array architecture and components

The systolic array operates by firstly computing the integral image for the incoming frame. Next, it computes the rectangles for each Haar feature in parallel for the image extracted region. Stage evaluation is also done in parallel for all locations in the search window, and after the outcome of each stage is known, the array proceeds by evaluating the next stage and its features in parallel over the entire search window. The candidate regions that fail each stage are marked and do not participate in the computation, in an attempt to reduce dynamic power consumption. Every CCU can act as the top-left-most corner for each feature, and is responsible for collecting the integral image values belonging to the rectangles for that particular feature. Each CCU holds the integral and integral squared image values, partial sums from rectangle and feature computations, and the variance for the search window that they represent as the top-left-most corner. Each CCU consists of minimal hardware to propagate data in all directions in the array, and is able to perform additions and subtractions, enabling the computation of the integral and integral squared image in a systolic manner. The rectangle sum can also be computed within the CCUs. The EUs are equipped with multiplexing hardware and contain a multiplier for stage evaluation purposes (to compute the weighted sum of each rectangle in each feature and the feature sum). A more detailed description of the units in the array is given next.

A. *Collection and Computation Unit (CCU)*

Each CCU represents the starting upper left corner of a search window in the image, and holds the necessary data for that window (such as image variance, whether or not the window

has so far passed the classification process, etc.). The CCUs are responsible for data movement throughout the system, collecting and accumulating integral image data for rectangle computation. Each unit is composed of an adder/subtractor, a local bus controller and a register file that holds the data necessary for the computation. The register file provides data storage for the integral image value, the squared integral image value, the collected rectangle sums (supports up to four rectangles per feature), the accumulated stage sum, the standard deviation of the image for the search window represented by that particular CCU, and temporary registers used to store data during data movement. Furthermore, the CCU holds a flag bit (FB) which is reset only when the search window represented by the CCU does not contain the object of interest. The bit is set at the beginning of every computation and is reset by the EUs at the end of a stage computation if the search window represented by that CCU does not pass a stage. To maintain temporal consistency, the bit is moved with the accumulated stage sum. A detailed block diagram of the CCU is shown in Figure 3-5. The CCU's critical path lies in the adder, depending on the required bit-width of the adder, various optimizations can be made to improve the speed of the CCUs.

Each CCU performs a set of predetermined actions. These are shifts to all four directions, addition and accumulation of incoming pixel values and squared pixel values, addition and accumulation of incoming rectangle points, and being idle. Each action is determined by the CUs, which send a global op-code of four bits to all CCUs, so that all CCUs can synchronize on the appropriate action.

One particular design parameter that the algorithm dictates lies on whether all CCUs should act as collection points for each feature. If each feature is evaluated over every possible location of the search window, then each CCU has to act as a collection and computation point. This is not always necessary however, it depends on the type and size of objects of interest. For example, a large object does not need a pixel-by-pixel exhaustive search, a search offset of five pixels will probably be sufficient. The modularity of the systolic array allows the designer to take advantage of this offset, by designing CCUs that are capable of collecting and computing rectangle information, and CCUs that do not. Additionally, CCUs located at the bottom and left parts of the array are not required to act as upper left collection points, since the feature computation will have reached the end of the search window. Therefore, a set of

CCUs can be designed without the adder/subtractor, and without the control logic necessary to perform collection and computation. These CCUs simply act as memory elements, and can be placed in locations in the array where the CCUs are not required to collect rectangle data.

An additional design optimization lies in the design of the adder inside each CCU and the registers holding the integral image and integral squared image values. As the location of each pixel in the integral image, relative to the integral image's origin increases, the value of the integral image (and the integral square image) increases in terms of required adder precision and in terms of storage requirements. This is illustrated in Figure 3-6(a). For example, for a 320×240 grayscale image (8-bits per pixel, maximum intensity of 255), the maximum value that needs to be stored at the bottom-right corner (*location 320,240*) of the integral image, is $320 \times 240 \times 255$ (in the unlikely event that all pixels have intensity value of 255). This requires 25 bits. Since the integral squared image is also needed, the bitwidth requirements increase to 33 bits for the adder. However, at location (20, 40), the maximum value that will be stored is $20 \times 40 \times 255$, which requires only 18 bits for the integral image, and 26 bits for the adder, to compute the integral squared image. Figure 3-6(b) shows the bit-width requirements of the adder, in a sample 320×240 image, to indicate a relative hardware demands as the location in the array changes.

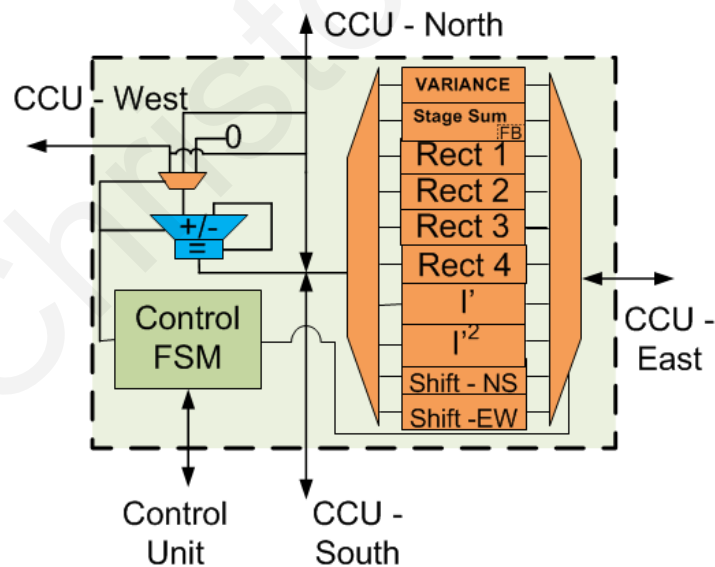


Figure 3-5. Collection and computation unit

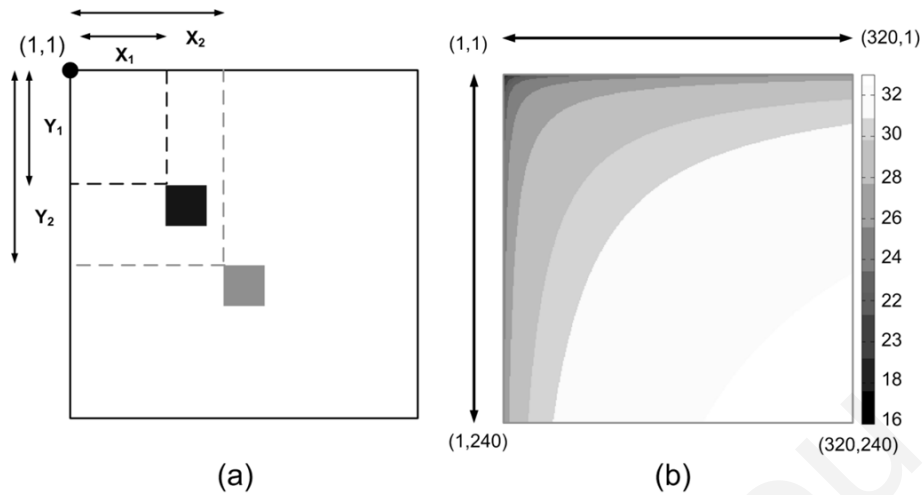


Figure 3-6. Processing array bitwidth requirements

Consequently, parameterized CCUs are designed, with variable adder bit-widths and variable-sized registers, which can be appropriately placed depending on the distance of each CCU relative to the origin of the array. This can be done either by one-by-one CCU case, or by designing different groups of CCUs with different bitwidths that can cover regions in the array, allowing some CCUs to have redundant bits. Alternatively, this can be done by limiting the size of the extracted image region partitions, this helps keeping the overall number of pixels required for both integral and integral square image summations small, resulting in relatively small bit-widths. The architecture was designed with the former approach.

B. Evaluation Unit (EU)

The EUs are located to the left of the array, at the beginning of each row, and act as input terminals to the array. The EUs are first used during the input of the search window into the array to compute the integral squared image values, by squaring each incoming pixel value to be used towards the squared integral image computation. During the computation, the EUs receive data from their neighboring CCUs (and through systolic manner, eventually from all CCUs in the corresponding row of each EU), starting from the rectangle values, the variance of the image and lastly the accumulated stage sum. The rectangle sums are multiplied with the rectangle weights read from the training set stored in off-chip memory. The variance is multiplied with the squared feature threshold, to determine the compensated threshold, which in turn is used to determine the feature sum to be added to the accumulated stage sum. If the

computed feature is the last feature of a stage, the accumulated stage sum is compared to the stage threshold and the flag bit is reset if the stage computation discards the search window. Else, both the accumulated stage sum and flag bit are shifted out using a toroidal link into the far right CCU. The EU starts the computation when signaled from the CUs, when the computation ends, the EU signals to the CUs to proceed with the next feature. The CU in the meantime stalls shifting in the CCU values during an EU computation, waiting on the EU to complete. When a stage is evaluated, the EU sends a signal to the CUs, so that they can coordinate all CCUs in starting the next stage of features.

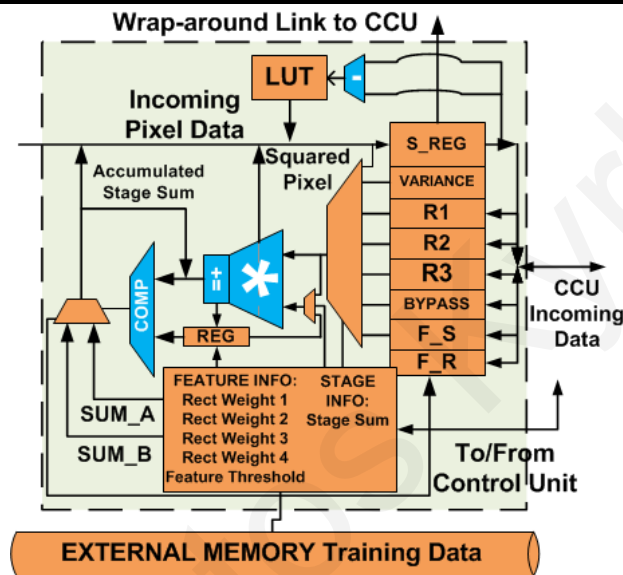


Figure 3-7. Evaluation unit

Each of the EUs interfaces to the external memory that holds the training data necessary for the feature computation, and reads the training data in a FIFO manner. Given that features are evaluated one at a time, the latency to retrieve the feature values does not affect the overall latency, as this can happen while the CCUs evaluate the feature's rectangles. This also enables storing of the training data (offsets) for all feature size sizes, thus removing the need to scale the rectangle offsets (d_x , d_y) dynamically when the computation shifts to larger feature sizes. It must be noted however, that feature scaling can be done on-chip as well, where each (d_x , d_y) offset can be scaled according to the preset feature scale factor. Upon computation of each feature, the next feature rectangle off-sets are read from the training memory and propagated along with the resulting feature sum, using the toroidal link, back to the CCUs. Consequently,

when a feature is evaluated with the rightmost CCUs receiving the last outcome of each computation, the entire array already holds the required off-sets for all rectangles associated with the next feature. The EU block diagram is shown in Figure 3-7. The EU multiplier has the longest critical path in the array, and various optimizations can be done to improve the frequency of the unit, such as using pipelined.

C. Control Unit (CU)

The systolic array architecture operation is controlled by distributed control units (CUs) which are spread in the system to reduce size of each CU's control region and wiring requirements. Each CU consists of finite-state-machine (FSM) logic, a multiply-accumulate unit, memory blocks and two control counters. One control counter is dedicated to the control flow while the other counts the number of objects that pass a stage, if no object passes a stage then it is possible to terminate the algorithm operation earlier. The FSM ensures that the computation units, the CCUs and EUs are synchronized to each other. The action they performed is determined via a four-bit op-code.

D. Systolic Computation Overview

The operation essentially is partitioned into the following stages: configuration, computation of integral and integral squared images, computation of image variance, computation of rectangles per Haar-feature, feature computation, stage evaluation and image evaluation. The computation is repeated for all upscaled features over a single search window, and for all search windows generated by the IPG. When the image has been searched at all search window sizes, the system is ready for the next image frame.

In each case, all units collaborate to perform the computation. Incoming pixels, stream in the array in parallel along all rows of the array, and are shifted in row-wise every cycle. The integral image and integral squared image are computed first. The computation consists of horizontal and vertical shifts and additions. Incoming pixels are shifted inside the array on each row. Depending on the current pixel column, each of the computation units performs one of three operations, it either adds the incoming pixel value into the stored sum, or propagates the incoming value to the next-in-row processing element while, either shifting and adding in the vertical dimension (downwards) the accumulated sum or simply doing nothing in the

vertical dimension. The computation is illustrated in Figure 3-8(a). To compute the squared integral image, the same procedure is followed. The incoming pixel passes through the multiplier in the EU, which computes the square of the pixel value, and then that value alternates with the original pixel value as inputs to the array. As such, the integral and squared integral image are computed in alternate cycles with the entire computation taking $2 \times [m + (m - 1) + (n - 1)]$ cycles, for an $n \times m$ input image.

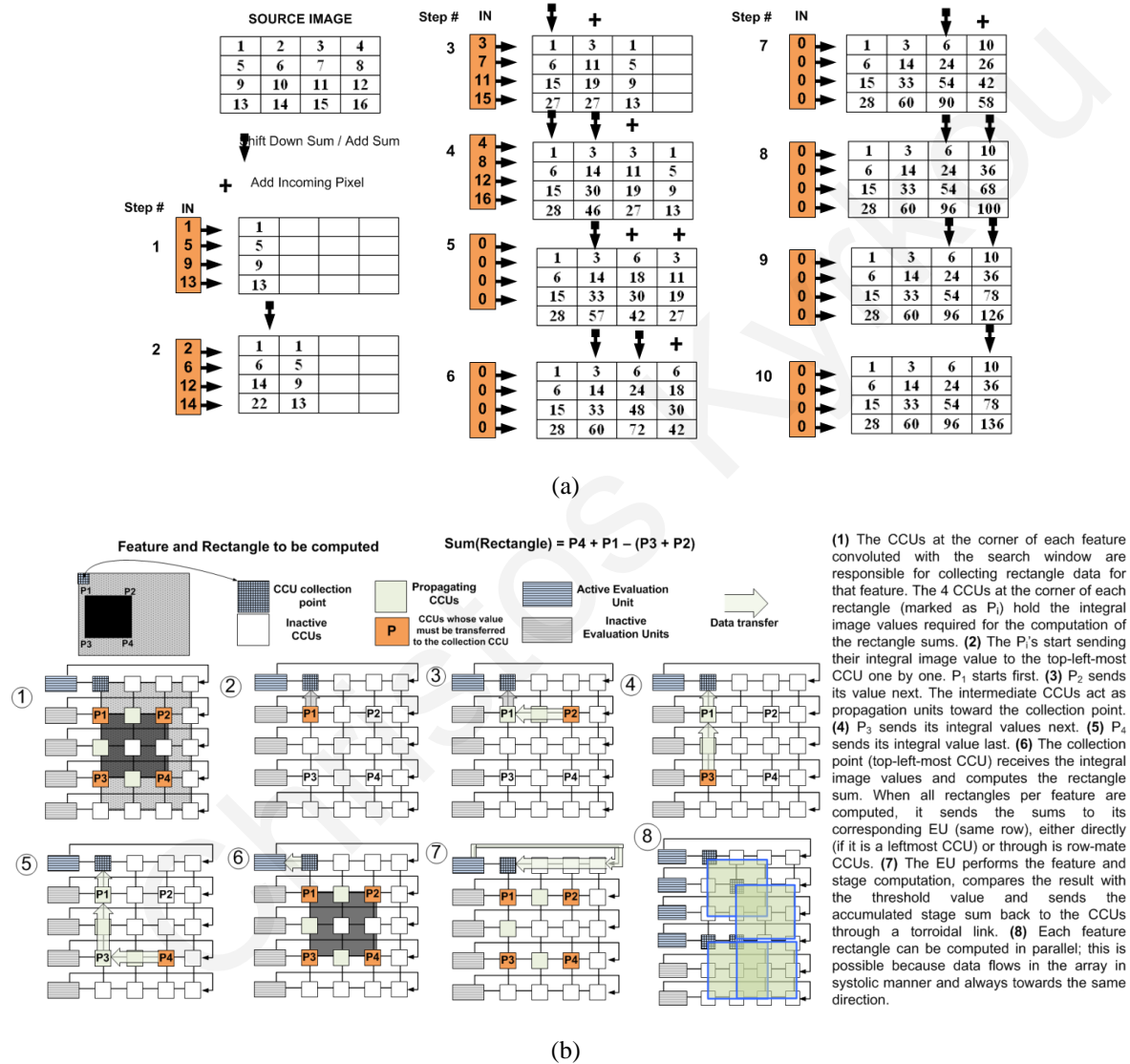


Figure 3-8. Systolic array computation flows

(a) Systolic array integral image computation and data flow (b) Systolic array feature computation flow

The rectangle computation takes place next. For each rectangle in a feature, each corner point is shifted towards the CCU acting as collection point. The points move one at a time, but in parallel for all rectangles in the array. At each collection point, the point is either added or subtracted to the accumulated rectangle sum, with the final rectangle value computed when all points of each rectangle arrive at the collection point. As such, each point requires d_x+d_y cycles to reach the collection point, where d_x and d_y are the offset coordinates of the point with respect to the upper left corner of the search window. When all rectangle sums for a single feature have been collected in the CCU that represents the starting corner for each feature, they are then shifted leftwards, towards the EUs, one sum at a time per EU. From left to right, eventually all sums arrive in each EU, where the rectangle weights are multiplied with the incoming sums, in order to evaluate the feature. It must be noted that each CCU contains the rectangle sums, the accumulated feature sum from the previous feature computation and the variance. Hence each CCU takes $n+2$ cycles to shift the data to its neighboring CCU, where n equals the number of rectangle sums per feature. When each rectangle sum enters the EU, it is multiplied with the respective rectangle weight given by the training set, and accumulated together to compute the feature sum. The compensated threshold is then computed using the original threshold and the variance as described earlier. The feature sum is then squared using the multiplier, and compared to the compensated threshold to set the feature result. The partial stage sum is accumulated with the feature result and shifted with the flag bit in a toroidal fashion to the CCU on the far right of the grid, to continue the computation. Eventually, when all feature results are computed, they are stored back into the CCUs in the grid and the computation resumes with the next feature. The computation is illustrated in Figure 3-8(b).

At the end of a stage, the computed stage sum is compared against the stage threshold obtained from the training set. Depending on the outcome of the comparison, the location of the CCU is flagged as an object of interest candidate and continues further evaluation, or is discarded by resetting the flag bit that is shifted with the stage sum. When a location which does not contain an object of interest arrives for computation at the EUs, the EU does not compute the feature sum, rather remains idle, and simply propagates the data to the far right CCU to resume computation. The CCUs that do not act as collection/computation points, (only hold integral image info as mentioned earlier) simply propagate their values in order to maintain temporal consistency.

When all stages complete for a certain feature scale inside the search window, the flagged locations correspond to the ones that contain the object of interest. If the feature computed is the last one, the computation ends. Each location that contains an object of interest is shifted to the right and outside of the array, for the host application to proceed.

This parallel approach yields several advantages. First, it does not require the training set data to be stored on-chip, as it only computes one feature at a time. Instead, it operates on the image data in parallel. Second, the systolic implementation returns predictable and fast operation, in the context of high frequency. Third, computations that are expensive in terms of delay and hardware overhead such as multiplication are isolated and computed together in parallel during each evaluation step, thus amortizing their delay towards the whole operation. Fourth, in contrast to the software implementation of the AdaBoost algorithm, the detection rate remains constant regardless of the number of objects of interest that exist in a single frame, whether they are detected positively or negatively (i.e. false positives). The software implementation suffers when the amount of object increases, as the algorithm will have to follow the entire classifier cascade multiple times. In contrast, the proposed architecture searches and performs the cascade only once for the entire image, rather than for each object, as done in software. Lastly, when used in conjunction with an IPG process, it can be scaled to the application's requirements and available budget, as the array size can vary from being equal to the size of the training feature to as much as the budget and performance requirements allow and demand.

3.4 Experimental Methodology and Evaluation Results

Two different approaches were followed to evaluate the architecture under different scenarios. First, the architecture was designed and verified using FPGA emulation. Second, a full functional simulation was performed using an ASIC implementation over a commercial CMOS library, with three different object detection applications used as benchmarks. Both systems were designed with emphasis on the corresponding hardware constraints, and evaluation was performed taking into consideration several design constraints and limitations. Prior to detailing the implementation details, the concept of performance under an object

detection system is first discussed, and list the factors, which explicitly impact the performance of the system.

3.4.1 Performance, metrics, limitations and constraints

There are two important performance metrics in object detection, the detection frame rate which defines the ability of the system to process a number of input image frames per second (FPS), and the detection accuracy, which defines the effectiveness of the system in detecting the object(s) of interest. For real-time video processing, the system needs to detect at least 30 FPS. However, if other image processing and recognition algorithms have to co-exist with detection, the system needs to be much faster, which is typically the case. Moreover, the system's accuracy largely depends on the training, and partially on the way that the training set is represented when implemented in hardware. In the development and design of the propose architecture, several performance metrics, limitations and constraints, were considered and are outlined in this section.

Firstly, the training set size, particularly the number of features and stages in the training set, largely impacts the performance. As each feature is processed in parallel, the algorithm depends on the total number of features and stages to return a positive result. Training set optimization can improve the performance by maintaining a high accuracy in the detection while reducing the number of features. This was done in [87], however, no detailed discussions were given related to the accuracy of the detector, especially the false positives. In the presented implementation the architecture is developed independent of the training set size, and thus can take advantage of potential emerging research that reduces training set data.

The second factor that has an impact on the performance is the size of the search window and the size of the array. As features are enlarged and computation is repeated to detect bigger objects, the number of enlargements necessary is defined by the search window size (i.e. until the size of the enlarged feature meets the search window size). Consequently, a large search window size will result in computation over several feature sizes. This increases the amount of computations and limits the performance. A small search window on the other hand limits the amount of feature enlargements and results in a larger number of search windows generated for each input image frame. However, the number of generated search windows increases

exponentially as the input image frame size increases, and potentially results in loss of data due to several downscaling iterations. The IPG and the systolic array are combined in the presented architecture in order to provide flexibility in selecting an appropriate search window size as the application demands, depending on the performance and cost requirements. The nature of the application, such as the amount of training data required, the feature size, the size of the object(s) of interest, input image size and number of objects concurrently appearing on the input image are all factors that play a role in determining a good ratio of the IPG to the search window size (and subsequently the systolic array size). A system-level optimization framework can potentially be used to determine these sizes for various applications, and is left as future work.

The third factor that impacts the performance is the input image frame size. Obviously a large frame results in more data to be explored and a larger number of search windows, but it also impacts the size of objects relative to the input image frame. Large-sized objects typically result in wasted computations when using small-sized features, whereas small objects result in wasted computations when using large-sized features. This is even worst when two or more objects of interest appear in different sizes, the largest the variance in the sizes of the objects, the larger the number of feature and stage computations overall. The proposed architecture is ideal for large image sizes, as the high degree of parallelism can process large images in parallel, resulting in satisfactory frame rates. Three image sizes were used for evaluation purposes indicating only a relatively small decline in the frame rate when going from a small to a much larger image frame size.

The fourth factor relates to the object of interest itself, and the targeted video application. In particular, the amount and size of objects of interest contained in a single frame plays a dominant role in the overall performance, especially in sequential software implementations. The big advantage of the AdaBoost algorithm, which results in large detection frame rates, lies in the ability of the algorithm to reject several search windows which do not pass certain thresholds during an early stage in the computation. However, if the amount of objects of interest in an image frame is relatively large, the algorithm slows down significantly, as it will have to go through the full computation several times. The architecture however, is independent of the number of objects, as the entire search window is explored in parallel, the

time required to search for a single object, is the same time as the time required to search for all objects in the search window. Furthermore, when two or more objects of interest of different sizes are present in the source image, detection will occur at different feature scales. A worst case scenario would be at for least one object of interest inside the search window, in every scale where a feature is evaluated. In reality however, this is a highly improbable scenario, a large object will usually cover smaller objects in an image. Furthermore, there are cases where objects of interest are not present in an image frame, the search windows will likely fail somewhere through the first few stages for all search windows at all feature sizes, thus enabling a new image frame to be processed. In such cases, the frame rate obviously increases. Additionally, changes within a video signal (i.e. new objects of interest entering the image frame or other objects leaving) typically happen within a few frames apart. Hence, consecutive frames are usually similar to each other. This of course implies that a lower frame rate than the video frame rate could be sufficient, however, in the likely scenario that object detection is part of a chain of operations that have to meet real time video processing, the detector still has to operate as fast as possible. Consequently, to conclusively evaluate any architecture, one has to choose a sequence of test frames containing a number of objects, of different sizes, taking these observations into consideration.

The last factor obviously lies on the hardware itself, most notably the operating frequency. In the design of the presented architecture, regular and modular components were used, with small critical paths. The CCU contains minimal hardware, with a fast carry-look-ahead adder. The EUs, which take more cycles, and burden potential delays in memory accesses and multiplications, only operate during certain time intervals (i.e. during each feature and stage evaluation and I/O operation), allowing the bulk of the computation (i.e. rectangle collection and summation) to the much faster CCUs.

3.4.2 FPGA Implementation and Emulation

A proof of concept of the proposed architecture, was designed targeting the Xilinx XUP Virtex II Pro platform [88]. The FPGA evaluation targets a face detection application, using the training set and parameters given with the Open Computer Vision library [18]. The Open CV library provides a state-of-the-art software implementation of the AdaBoost detector,

utilizing a very accurate pool of features. The training set uses a starting feature size of 24×24 pixels, and scales each feature by a factor of 1.25 (taking the ceiling of the result), resulting in 5 scaled feature sizes (24×24 , 30×30 , 38×38 , 48×48 and 60×60). Each feature has between 2 to 4 rectangles. The training set consists of 2,913 features in 25 stages. The number of features per stage range from 9 to 211, and the total number of rectangles is 6,383.

The training set is organized on a feature by feature basis. Each feature data includes the feature sequence number, the number of rectangles in the feature, the d_x and d_y offset values for each rectangle in the feature, the weight associated with each rectangle, and the squared threshold value for each feature. Additionally the stage data includes a certain threshold per stage. To represent the training set, 8 bits per rectangle weight were used, for each threshold value and for each predetermined feature sum, using signed fixed point numbers of 2 integer bits and 5 decimal bits. The dynamic range supported is ± 3.96875 , close to the required accuracy for the Open CV training set. The external memory that holds the training set, holds also the upscaled feature data, that is rectangle offsets and weights. The rectangle offset was stored using 6 bits each, as the largest feature size utilized was 60×60 (to fit the 80×60 array). Each rectangle needs to store up to 4 d_x and d_y values. The training set is stored in the off-chip (on-board) memory, as features and stage data are used only once every array collection and computation. As already mentioned, the upscaled feature data were stored in off-chip memory as well, since when features are enlarged, new rectangle offsets are used. This however is of minor importance, as the offsets can simply be scaled on-chip, dynamically, since the feature training set is loaded feature by feature. For simplicity purposes, the rectangle offsets for all feature sizes are stored in off-chip memory, as part of the training set.

In designing the CCUs, adequate storage needs to be provided for the case where all pixels will have an intensity value of 255, an unlikely scenario, but necessary for correct operation. Thus, the maximum integer value that can be stored in an integral image is $255 \times 80 \times 60$ and the maximum integer value that can be stored in an integral squared image is $255^2 \times 80 \times 60$. This requires 21 bits and 29 bits respectively. Knowing these requirements, the architecture was designed using 80×60 CCUs, 60 MEUs and 4 CUs. Each CCU connects to

its neighbors through an 8 bit bus, which however can increase to a larger size if necessary for bandwidth purposes. The platform contains external memory (DRAM), which was used to store input image frames and the training set. The evaluation image were landscape grayscale images of size 320×240 pixels, and an array size of 80×60 cells (60 rows by 80 columns), the largest size that could fit on the targeted FPGA, maintaining the 4:3 ratio of the initial image frame). The IPG receives 8 pixels per clock cycle (the DRAM I/O bandwidth), and generates 80×60 sized search windows, at a pixel offset of five pixels (i.e. every search window starts five pixels after the previous). The IPG also downscales the image by scale factors of 0.75 and 0.5, creating three downscaled images of sizes 240×180 , 160×120 and 80×60 . The generated downscaled images are stored on the FPGA Block RAM. The number of downscaled images is parametrizable, the scale factor is simply stored in a register, and scaling is done by matrix multiplication. The IPG uses two 80×60 frame buffers, generating search windows in lockstep fashion (i.e. it generates the first, and then proceeds to generate the second while the systolic array processes the first one, with both the IPG and the array alternating between each buffer). FPGA synthesis and utilization results were collected using the Xilinx ISE 9.2 and are shown in Table 3-2.

The system, which system operates at 100 MHz, was verified and evaluated using the application of face detection, through a sample of 300 test images which contained several faces of different sizes, obtained through the World Wide Web, and sized and formatted to the design requirements. The test images were stored in a Compact Flash card during the system initialization stage, and then loaded on the DRAM prior to running the detection framework.

TABLE 3-2. SYNTHESIS RESULTS FOR THE VIRTEX II PRO FPGA IMPLEMENTATION

FPGA Resources	Slices (13696)	Flip Flops (27392)	LUTs (27392)	BlockRAM (312.5kB)	Multipliers DSPs (136)
<i>IPG</i>	2,248 (16%)	2,101 (8%)	2,445 (9%)	52.8kB (17%)	8 (5%)
<i>Array (80x60)</i>	10918 (80%)	20185 (74%)	22706 (83%)	0 (0%)	60 (44%)
<i>System</i>	13455 (98%)	23744 (87%)	25118 (92%)	52.8kB (17%)	68 (50%)

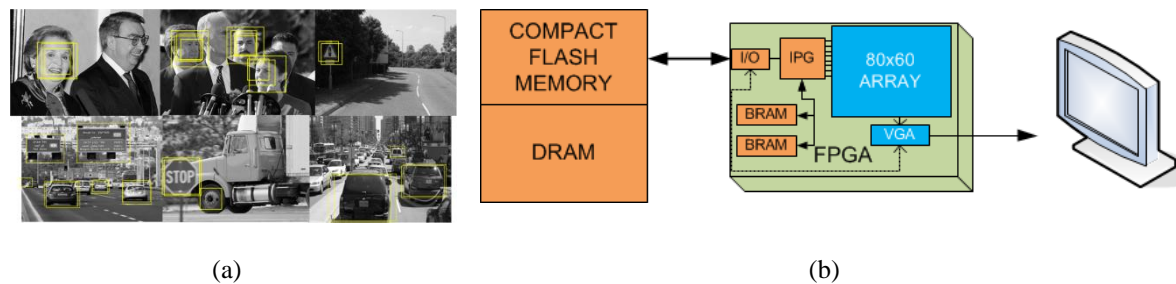


Figure 3-9. FPGA system prototype

(a) Output image frames (b) Experimental platform block diagram

The frames were input to the detector, which processed them. A custom VGA controller was then designed and used in order to output the result of the detector to a VGA monitor, for visual verification, along with markings on where the candidate faces were detected. A diagram of the FPGA prototype and image detection results are shown in Figure 3-9.

The frame rate depends on several factors, some of which are independent of the architecture. The system processed all 300 test images in 4.68 seconds, an estimated rate of 64 frames per second, which for the type and frequency of the FPGA is relatively high. Additionally, the FPGA implementation achieved 93% accuracy detecting the faces on the images when compared to the corresponding Open CV software implementation, running on the same test images. This discrepancy can be justified to the fact that the Open CV implementation only scales the features up (no image downscaling) which does not result in data loss. Additionally, some training data was not able to be represented within the dynamic range employed by the hardware design.

Table 3-3 presents a comparison table between existing FPGA implementations along with their characteristics, and the proposed architecture implemented on FPGA, for the application of face detection. As seen in the table, the proposed architecture is significantly faster and more accurate than most implementations. The implementation in [86] was based on pure cycle-accurate simulation rather than implementation, and the clock frequency is twice as the one used in the evaluated implementation (which was limited by the capabilities of the FPGA board). The work in [87] yields a reported 143 FPS, but uses a much smaller training set (two orders of magnitude smaller than the one in Open CV) in order to achieve such a high frame rate. Furthermore, the reported accuracy focuses on a specific data set, and no detailed

discussion is provided on the rate of false positives. The implementation in [7] uses a 80×60 image, an order of magnitude less features, and two search reduction methods (skin and motion detection) in order to achieve over 600 FPS, however it only considers three object sizes all these optimizations considerably minimize the size of search windows that need to be processed. In addition no accuracy information is provided to measure the impact of all these optimizations.

TABLE 3-3. RESULTS COMPARISON OF RELATED WORK IMPLEMENTATIONS ON FPGAS

Work	Hiromoto [83]	Cho [84]	Wei [85]	Shi [86] ^a	Lai [87]	He [7]	Presented Architecture
FPGA	XC5VLX3 30-2	XC5VLX1 10	XC2V2000	N/A	XC2VP30	XC5FX130 T	XC2VP30
Frames per Second	30	26	15	102	143	625	64
Image Size	640×480	320×240	120×120	176×144	640×480	80×60	320×240
Cascade Information	Stages	25	22	3	25	1	25
	Features	2913	2135	225	2913	52	2913
FPGA Resources	Slice LUTs	63443 /207360	66851 /69120	13853 /21504	N/A	20901 /27392	25818 /27392
	Slice Registers	55515 /207360	21,902 /69120	4573 /21504	N/A	7782 /27392	23744 /27392
	DSPs	N/P	N/P	28/56	N/A	N/P	68/136
	Mem	N/P	41/128	56/56	N/A	44/136	24/136
Clock Frequency (MHz)	160.9	N/P	91	200	126	73	100
Accuracy	N/P	N/P	85%	N/P	86% on faces	N/P	93% (overall)

^a Implementation of a cycle accurate simulator
N/P – Not Provided | N/A – Not Applicable

3.4.3 ASIC Implementation and Evaluation

In addition to the FPGA emulation, a larger system was designed (that could not fit on large existing FPGAs), targeting an ASIC implementation. The objective of the ASIC implementation was to obtain experimental insights on the scalability and feasibility of the proposed architecture, towards large scale integration. The ASIC implementation was evaluated using three object detection applications, face detection, road sign detection, and vehicle detection. The training set used in the face detection application was the same one used in the FPGA prototype. The training set for the other two applications was constructed using Open CV and MATLAB, using sample images obtained from the World Wide Web. The

TABLE 3-4: DETECTION APPLICATIONS TRAINING DATA

Detection Application	Object Per Frame	Feature Size (pixels)	# of Rectangles per feature	# of Stages	Total # of Features	Detection Accuracy
Face	1-7	24×24	2 to 4	25	2913	93%
Road Sign	1-2	12×12	2 to 3	12	414	83%
Vehicle	2-10	24×36	2 to 4	20	1715	78%

objective was not to construct an accurate training set per se, rather than a realistic one to be used as an experimental set. The training sets were constructed using road sign and vehicle images, and training set details for each application (including the face detection) are given in Table 3-4. Input images of four sizes (1024×768 , 800×600 , 640×480 and 320×240) were used, again obtained through the World Wide Web, containing several faces, road signs and vehicles, depending on the targeted application. Then proceeded to design and implement an architecture which could receive as input at least a 1024×768 grayscale image, and process it as fast as possible, using the training sets mentioned. It must be noted that each application differs from each other in the context of their training sets (and feature sizes), the underlying hardware architecture is the same for all the targeted applications, as well as input image sizes and formats.

The experimental platform was designed using search windows of size 320×240 pixels. Consequently, the size of the array was set to be the same, consisting of 320×240 CCUs and 240 EUs. The IPG was designed with two image partition buffers, producing search windows in similar fashion to the FPGA implementation. The original input image size was scaled down using a scale factor of 0.75, and the features were scaled up using a scale factor of 1.33. The training set, downscaled images from the IPG and input image frames were modeled as external memory, everything else was considered on-chip. Additionally, all parameters outlined in the FPGA implementation were modified to reflect the new search window size (such as storage considerations for the integral and integral squared images, data bus between CCUs and EUs, etc.).

The system was synthesized with using Synopsys Design Compiler targeting a commercial TSMC 65nm CMOS library, in order to obtain relevant metrics such as area, operating frequency and power consumption. The default library values, and Synopsys' synthesis

primitives were used (focused on area optimization over performance), as well as components from Synopsys Designware IP library. Pre-layout results indicated that the critical path in the system was identified in the EU multiplier (a 64-bit multiplier). An 8-stage multiplier from Designware IP library was used in the design targeting high frequency. It must be noted that the synthesized design does not consider the IPG memory modules, hence the CACTI toolset [89] was used to obtain the potential operating frequency for the two IPG memory modules, estimated at 800 MHz. As such, the targeted frequency of the entire system was set to 800 MHz. The post-synthesis, pre-layout results also indicate an area estimation of roughly 88 million transistors.

Preliminary results also were collected for some indicative power consumption merits using 1V power supply voltage and 50% probability of switching activity on all lines. Prior to reporting the obtained power consumption results however, it needs to be stated that the overall power consumption depends on several factors not related only to the architecture. The power consumption depends on the input image size and subsequently the number of downsampled images produced, the number of search windows, the number of features in the training set and the number of necessary computations. The latter is determined by the number of objects of interest found in the input frame. Obviously the chosen operating frequency and power supply of the system are important as well. Consequently, power comparison with architectures found in literature is not suitable without the use of the same input data sets and input image sizes. Hence, instead of reporting only the total power consumption for one frame, an overview is given of how this power is consumed throughout the computation.

To compute and evaluate a 24×24 feature (rectangle collection and computation, propagation to the EUs, computation and evaluation in the EUs, and propagation of the feature sum back to the array), the system consumes 0.06mW. The IPG unit also consumes ~ 0.014 mW to downscale a 320×240 image to a 240×180 image and 0.0023 mW to produce one 80×60 search window. Overall, to compute a single 320×240 frame with one object of interest (human face), the unit consumes ~ 2.45 mW of power. The only power optimization mechanisms considered was to not compute feature values when a region was marked as a non-candidate since the overall focus was not on optimizing with regards to power.

TABLE 3-5: ASIC IMPLEMENTATION - SIMULATION RESULTS

Application /Accuracy	Detection Accuracy	Input Image Size	Time to process 10 frames (seconds)	Projected Frame Rate (FPS)
Face Detection	95-96%	1024×768	0.109	~91
		800×600	0.098	~102
		640×480	0.084	~118
		320×240	0.075	~133
Road Sign Detection	92-97%	1024×768	0.099	~101
		800×600	0.093	~107
		640×480	0.083	~120
		320×240	0.072	~139
Vehicle Detection	91-96%	1024×768	0.128	~78
		800×600	0.116	~86
		640×480	0.108	~93
		320×240	0.098	~102

Using text files that contained the input image files and training set data, functional RTL simulation of the system was performed using Modelsim using a set of 10 test images per image size per application (i.e. 40 test images per each application), and obtained the total number of cycles required to process each test case. The resulting frames were stored as text files, and reconstructed to images using MATLAB, to visually verify the results. Using the obtained clock frequency from the synthesis results, the detection frame-rate was estimated (as well as the detection accuracy when compared to the corresponding software implementation). Table 3-5 summarizes the results for each application, under the four input image frame sizes.

Table 3-6 presents a summary of the synthesis results, and a brief comparison with the special-purpose vision processor presented in [81]. When comparing equal sized input images, the frame rate achieved by the proposed architecture is significantly larger. The associated power consumption and hardware overhead costs cannot be compared, however, the overall simulation results indicate that the proposed architecture can be scaled to significant sizes, and potentially be used in high-performance applications with large input image sizes, or can be designed to consume minimal energy and hardware overheads for small-scale embedded systems.

TABLE 3-6: ASIC IMPLEMENTATION - RELATED WORK COMPARISON

Work	Proposed Architecture	Hanai [81]
Technology	TSMC 65nm CMOS	90 nm CMOS
FPS (320x240)	~133	8
Area (# of transistors)	88 million	2.1 million
Power (mW)	2.45 per 320x240 frame	0.47/FPS – 3.72 total
Clock Frequency	800 MHz	54 Mhz
Accuracy	95%	81%

3.4.4 Discussion

Both the FPGA prototype as well as the large scale ASIC implementation have shown great potential for applications with real-time performance requirements, such as real time object detection in vehicular embedded and applications involving multiple camera streams. The system is particularly useful in monitoring populated areas such as airports and transportation terminals, where it can process frames from alternate video streams, regardless of the amount and size of objects found in the input image frames. The scalability of the system and its independence from the training set also provide flexibility to the designer, allowing the designer to determine the most efficient size of the system directly from the application requirements. By merging the IPG with the feature upscaling originally used, the system achieves a fully parametrizable performance-to-cost ratio, if the silicon budget allows it, an increase in the array size will boost the performance (by increasing the degree of parallelism). On the other hand, a smaller array, while slower, costs less and can still satisfy certain performance requirements.

There are some useful conclusions extracted from the simulations with respect to the algorithm. The FPGA implementation shows that the architecture can scale well in smaller, less demanding environments, while maintaining reasonable frame rates. The ASIC implementation on the other hand illustrates the full-throttle operation of the detector, and its suitability for multiple video streams and detection of objects that could appear in different numbers and sizes within an input image frame. Obviously, depending on the budget and

application constraints, the designer can select the type of implementation that satisfies the operating conditions and application specifications.

3.5 Conclusion

Object detection is an important step in multiple applications related to computer vision and image processing, and real-time detection is critical in several domains. This chapter presented a systolic-array inspired accelerator for the AdaBoost-based Haar feature cascade classifier proposed by Viola and Jones that can perform generic object detection using the AdaBoost algorithm. The architecture targets both parallel computation and parallel data movement and thus combines an image pyramid generation process, along with highly parallel systolic computation, to offer a flexible design that is suitable for several types of applications and hardware budgets. These features allow the architecture to process the input image in a radically different manner than other works by parallelizing the processing of multiple windows instead of just one and to also permit for feature upscaling to be performed. Furthermore, the advantage of the architecture over software implementations is that it maintains a constant frame rate regardless of the number of objects found in the image. Two experimental platforms of the architecture were presented, a low-end FPGA implementation and a high-end ASIC implementation, both of which achieved significantly high detection frame rates and accuracy compared to related works.

Further optimizations in terms of power consumption can significantly improve the architecture. System-level optimization algorithms, of determining a systolic array size that best satisfies the performance/cost requirements can be explored. Finally, this architecture can be combined with other on-chip implementations of related applications to form a complete high-performance embedded computer vision and image processing hardware platform.

The seminal work by Viola and Jones has been considered a major step towards real-time object detection. One drawback of this approach however, is that detection performance depends heavily upon the available pool of features which the AdaBoost uses to specify weak and strong classifiers [90]. Alternative algorithms such as Support Vector Machines offer built-in mechanisms for new feature generation and can often perform equally well [3],[91].

Hence, in addition to the proposed Haar-cascade architecture, the hardware acceleration of Support Vector Machines (SVMs) is also considered and is presented in Chapter 4.

Christos Kyrkou

Christos Kyrkou

CHAPTER 4

HARDWARE ACCELERATION OF SUPPORT VECTOR MACHINES

Support Vector Machines (SVMs) [39] have emerged as a powerful supervised machine learning algorithm which has been widely adopted since its introduction by Cortes and Vapnik [16]. SVMs are considered a very good tool for classification and regression applications due to their mathematical formulation which is based on statistical learning theory [39] and guarantees good generalization capabilities. In particular their major advantage over other methods such as neural networks and Gaussian mixture models is that they are less likely to get stuck at a local minimum during training [92] and also the structure of the classifier is data-driven and thus determined automatically [93]. Due to these properties SVMs have exhibited excellent classification accuracy rates for a wide range of applications [19], [93],[94]. Consequently, they have attracted a lot of interest from the computer vision community to be used in object detection and image processing applications such as pedestrian detection [95],[96], car side view detection [97] and face detection in [76], [98], [99] and have again demonstrated high classification accuracies. However, the major disadvantage of SVMs is their slow classification speed. The classification complexity of SVMs is proportional to the number of training samples needed to specify the separating hyperplane between classes, which are called support vectors (SVs). Hence, for large scale problems, the high classification accuracy rates demonstrated by SVMs come at the cost of increased computational complexity. As such, when considering embedded applications, e.g. embedded vision, automotive, and security, SVM-based classification systems with hundreds of support vectors and a large number of instances that need to be classified, find it difficult to meet real-time and power consumption constraints under limited resources and area constraints.

4.1 Related work on Acceleration of SVMs

Dedicated SVM hardware architectures have emerged as a potential solution in order to meet real-time performance and power-consumption constraints in embedded applications. The majority of related work on SVMs can mostly be divided into three categories: (i)

application specific architectures that are tailored to specific classification problems (ii) optimizations that aim at reducing the hardware complexity and improve classification speed and (iii) coprocessors for speedup of the training phase. While there has been a considerable amount of work done in accelerating support vector machines with dedicated hardware, there is a lack of works that have looked into hardware architectures for embedded object detection that are not constrained by the vector dimensionality and can be scaled to processed multiple windows from an input image. Furthermore, only recently there has been some work on trying to accelerate cascade SVMs. A detailed description of related works on SVM accelerators is given next.

There exists a fair amount of work on accelerating both the SVM training and classification for general-purpose processors and DSPs, aiming to provide higher performance on such platforms. The work in [100] presents an evaluation of SVM implementation on embedded processor architectures, and proposes architectural modifications in order to improve their performance. An analysis was performed in [101] where critical parts of the SVM algorithm were mapped between hardware and software, demonstrating how hardware can be used to accelerate SVM computations. An implementation of an SVM classifier on a microcontroller was presented in [102]¹, dealing with issues such as limited memory and hardware. Attempts to accelerate the SVM training phase with multi-core processing platforms have been demonstrated in [103]². However, performance on such systems is limited by locality issues and limited cache size for large dimensionality vectors. Furthermore, multi-core processing systems require higher power consumption than dedicated hardware accelerators. Overall, the limited resources of DSP and multiprocessor systems does not provide the necessary parallelism to allow for real-time operation.

NVidia's compute unified device architecture (CUDA) has also been used in [104]³, [105]⁴, [106]⁵, [107]⁶ in order to speedup SVM classification using the parallel

¹ Microcontroller: PIC16F877 @ 1MHz

² 2 CPUs 8 Cores @ 2.3 GHz, 4GB RAM

³ Intel Core i7 920 @ 2.67, 6GB RAM | NVIDIA Tesla C1060 @ 1.3 GH, 4GB RAM

⁴ GeForce 8800 GTX @ 1.35 GHz. 768 MB RAM

⁵ Intel Core 2 DUO @ 2.66 GHz | NVIDIA GTX260

⁶ Intel Core i7 CPU 920 @ 2.66 GHz | NVIDIA GeForce GTX 925

computing resources of a GPU showing improved results compared to CPU implementations. However, GPUs are power hungry devices compared to FPGAs [108],[109],[26], as FPGAs consume approximately an order of magnitude less power, and as such they are not suitable for power-constrained embedded applications. In addition, existing GPU implementations do not translate well to the more energy-efficient embedded GPUs due to less available resources (less registers, cache, cores) [12].

Hardware implementations of SVMs been also proposed both for the training as well as for the classification phase, in order to address the constraints and limitations of the above platforms using custom hardware architectures, while mostly utilizing FPGAs. Early work on the hardware implementation of SVMs focused on simple circuitual architectures for the training phase [110]. More recently, other works [111] for the SVM training phase exploit the diversity in bitwidth precision requirements of the training data in order to develop scalable architectures. Architectures have also been developed for the SVM classification phase in order to design intelligent embedded systems. A mixed-signal SVM processor was presented in [112], utilizing analog computation for accuracy and digital output for VLSI integration. Anguita et al [113] propose hardware architecture for a modified SVM training algorithm showing comparable results with respect to the widely used Sequential Minimal Optimization (SMO) training algorithm [45]. Implementations of SVM classification systems have been restricted to small scale data [114], with only a few support vectors [115],[116] low dimensionality [115],[117],[92],[118],[119] and small-scale multiclass implementations [120],[121],[122]. Overall, the architectures proposed in these works use parallel processing units to rapidly process a single vector or multiple processing units to process many support vectors in parallel for a single input. In both cases, however, the proposed architectures were developed for specific problems and thus are not easily extendable to other scenarios and cannot be easily scaled to process multiple windows or support vectors simultaneously. The works in [108] and [123] have tried to develop coprocessors for CPU clusters in order to speedup SVM computations. They utilize clusters of vector processing elements and also compare the proposed hardware implementation with GPU and CPU SVM implementations, demonstrating both higher performance and lower power consumption. However, these works offload the kernel computations to the CPU coprocessor and the parallel processing capabilities depend on parallel input through the PCI express and external DRAM which have

high power consumption and are thus unsuitable for embedded applications. Nevertheless, considering the implementation of kernel functions is necessary for standalone embedded visual object detection systems. From the aforementioned works only [107],[116],[119] use SVMs for visual object detection applications utilizing FPGAs.

The main vector operations in SVMs require multiplications that can be expensive in terms of area, power, and performance. Thus, research has also been done on potential simplifications to make SVM classification and training more suitable for digital implementation and suitable for devices with limited computational resources. One of the earliest attempts was proposed by Anguita et al [124], where the multiplications in the training phase are substituted by shift and add operations. Other approaches use training algorithms commonly used to train neural network, and are more suitable for hardware implementation, and adapt them to train SVMs [125]. Similar approaches have also been proposed for the SVM classification as well. These approaches include using CORDIC algorithms to compute kernel functions [117],[121],[122], however, more compact implementations that require less hardware have increase latency [122], thus the experiments were limited to problems with a small number of input data and SVs. Other works [126],[127] proposed that the computations be done in the logarithmic number system so that all multiplications are substituted by additions. This approach requires a costly double transformation from the decimal number system to the logarithmic one and back again in order to compute a multiply-accumulate operation. However, they only consider a single processing module, hence, when adopting a more parallel architecture, to facilitate real-time operation, the additional cost from incorporating the double transformation, for all the inputs, to convert between the decimal number system to the logarithmic one and back again increases. Alternatively, a pseudo-logarithmic number system was proposed in [128], however, the overheads for converting between number systems in order to perform additions still remain. The works in [129], [130], [131],[132], have looked at how the bitwidth precision impacts the classification error, in an effort to find the best trade-off between hardware resources, performance and classification speed. Although the kernel operations still need to be implemented with multipliers leading to high resource demands for parallel implementations. A hardware friendly kernel was proposed in [133] as an alternative to the RBF kernel. It operates in conjunction with a CORDIC

algorithm to remove multiplication operations, however, it is unclear how this kernel behaves for a variety of applications and as such may not have the generalization capabilities of more established kernels. Finally, in [134], Irick et al propose hardware optimizations for the RBF kernel.

The aforementioned works have only considered hardware implementations and optimizations for monolithic SVMs while hardware, implementations for cascade SVMs have only recently been explored. In [135] and [109] the authors implement cascade classifiers with low and high precision bitwidth and also explore word-length optimizations for heterogeneous datasets. The dynamic ranges of heterogeneous datasets are exploited to design a cascade architecture for SVM processing that is optimized with respect to the bitwidth precision and throughput.

Considering implementations of previous works for SVM there is lack of architectures that can provide efficient parallel processing for both support vectors as well as multiple windows in a scalable manner that is suitable for object detection applications. In addition related works for SVM classification consider the implementation of a single kernel and the architecture is not generic to handle different kernel operations. Furthermore in their majority, none the previously presented works focus on the implementation of architectures for cascade SVMs, and are instead only optimized for monolithic SVM processing. Moving towards large scale embedded applications where thousand windows need to be classified, cascade SVMs will need to be utilized to provide speedups. Introducing new challenges for hardware implementation since cascade SVMs multiple classifiers need to be implemented on the same silicon fabric, each with different processing and memory requirements. As such, single SVM architectures, which do not exploit the properties of the cascade classification scheme, are not suited for this purpose. Hence, this research addresses the above two issues by proposing two architectures one for generic and parallel SVM processing and one for hardware-efficient cascade SVM processing. The following two sections detail the architectures and experimental evaluations for monolithic as well as cascade SVM architectures.

4.2 Hardware Acceleration of Monolithic SVMs

Embedded visual object detection applications require processing thousands of windows with hundreds of support vectors [76], [136] in real-time. As the number of input vectors and vector dimensionality increases existing architectures that rely on rapid processing of a single vector will not scale since there will be increasing demand for processing resources as well as higher power consumption. In addition other architectures that process multiple support vectors do not address the kernel implementation. To tackle these issues a flexible architecture is needed that is constraint by the vector dimensionality and can scale depending on the number of support vectors and input windows. Hence, an array processing architecture is presented that addresses the above concerns. The array-based architecture has many benefits compared to previous works, as it provides parallel processing of many input and support vectors, it is modular and regular thus allowing for a scalable design and reduced complexity, as well as efficient memory management and data flow between the processing units. Furthermore, the hardware units are designed to implement different kernels thus the same architecture can be used for different object detection applications, while demanding hardware units are shared amongst between the more common units. Finally, the proposed architecture is flexible to the application characteristics since it can be modified to allow for single input vector processing with multiple support vectors, multiple vector processing with groups of support vectors and can also be configured to handle multi-class problems, which has not been considered in previous works.

The array processing architecture is comprised of three main regions. The memory region comprised of a chain of memory units where the support vectors and alpha coefficients are stored, the vector processing region which is responsible for the vector processing, and is the largest region in the array, and the scalar region that processes the results produced from the vector operations. The array is comprised of two types of processing elements that serve different purposes: the Vector Unit (VU) is used for all vector computations, and the Scalar Unit (SU) operates on the scalar values produced by the VUs. In addition to these processing elements, the architecture contains dedicated memory units that feed the array with training data, and a finite state machine (FSM) control unit that synchronizes the array operation. The structure of the array is shown in Figure 4-1. The main regions of the array are detailed next,

followed by a description of the array data flow and the issues associated with the scalability of the array, as well as details on extending its operation to address multi-class classification problems.

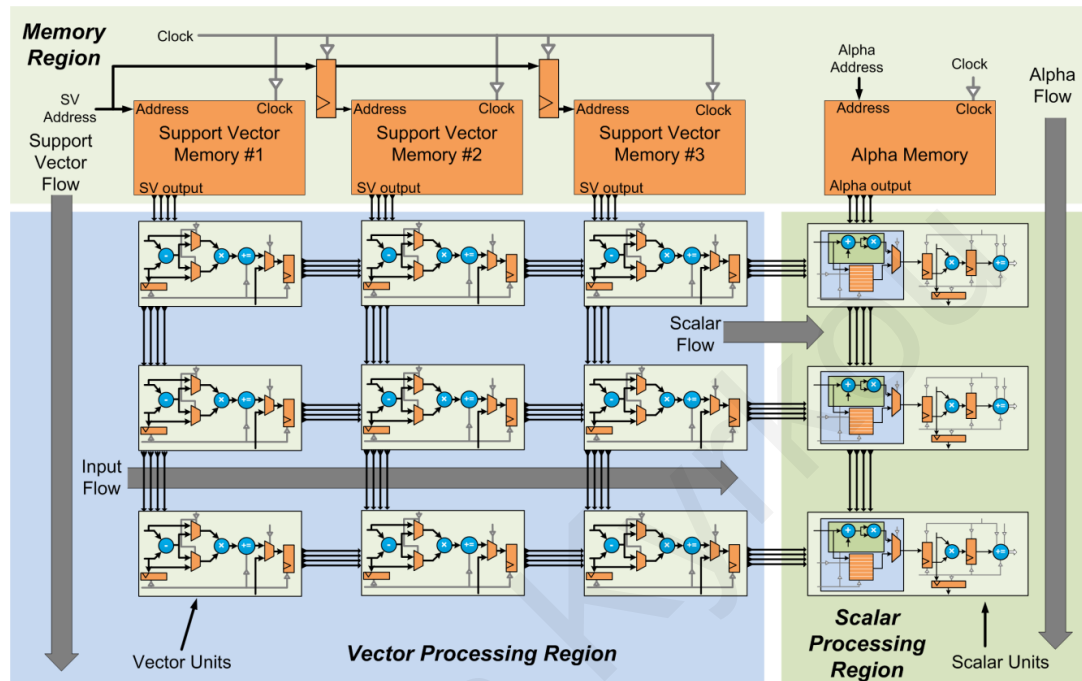


Figure 4-1. Support vector machine array processing architecture

4.2.1 Array Processing Hardware Architecture

A. Vector Processing Region

The Array is comprised mostly of VUs (shown in Figure 4-2) which are responsible for processing input windows with support vectors according to the kernel function operation to produce the scalar values required for the latter computations. Multiple VUs are interconnected to form a processing array to allow for massively parallel vector processing. A generic VU should be able to perform the necessary operations to process the most common kernels (shown in Table 4-1). These operations are the dot product $x_i \cdot x_j$ for linear and polynomial kernels and the Euclidean norm $\|x_i - x_j\|$ for the RBF kernel, which happen on each component of vectors x_i and x_j . The latter requires the multiplication and accumulation of vector components while the latter requires the squaring and accumulation of the difference

of two vectors. In [108], the dot product variant of the RBF kernel is used resulting in a more uniform architecture that computes just dot products. However, the variant is not as numerically stable and also requires that squares of vectors are pre-computed and stored on

TABLE 4-1 COMMON SVM KERNEL FUNCTIONS

Linear: $K(x_i, x_j) = (x_i \cdot x_j)$	Equation 4-1
Polynomial: $K(x_i, x_j) = ((x_i \cdot x_j) + \text{const})^{\text{degree}}, \text{const} \geq 0, \text{degree} \in \mathbb{N}$	Equation 4-2
Homogeneous Polynomial: $K(x_i, x_j) = ((x_i \cdot x_j))^{\text{degree}}, \text{degree} \in \mathbb{N}$	Equation 4-3
RBF: $K(x_i, x_j) = \exp\left(-\frac{\ x_i - x_j\ ^2}{2 \times \text{sigma}^2}\right), \text{sigma} > 0$	Equation 4-4

chip which dramatically increases memory demands. Thus the initial RBF kernel is considered for the implementation of the architecture. However, instead of designing different components for each kernel the components necessary for the implementation of the RBF- and dot-product-based kernels have been merged into a unified configurable architecture. In this way the RBF kernel only requires minimal additional hardware in the form of a subtractor, compared to the dot product-based kernel. Hence, the VUs are comprised of a subtractor, a multiplier, an accumulator and multiplexing logic to satisfy the processing needs for both vector operations. The multiplier either computes the product of the two vector components, or squares the difference of the two vector components. An 8×8-bit multiplier would suffice for the majority of object detection applications most of which operate on grayscale images (8-bits per pixel). The result of the 8×8-bit multiplier (a 16-bit value) is passed to an accumulator to complete the dot product computation. The bit-width of the accumulations is proportional to the vector dimensionality, denoted as c . The accumulator therefore, performs c accumulations of 16-bit operands, thus the accumulator bit-width requirements are given by $\log_2(c \times (2^{16} - 1))$.

The VUs are interconnected with each other through a vertical and a horizontal pipeline. The Support vectors travel through the vertical pipeline while the input vector data and scalar results travel through the horizontal pipeline. Each VU has three operational states: PROCESSING, IDLE and TRANSFERRING. In the PROCESSING state the VU is involved

with the computation of the vector operation, and simultaneously transfers data/control values to its neighboring VUs. During the TRANSFERRING state, the scalar value computed by each VU is transferred towards the SUs. Transfer data switching is done through a 2-1 multiplexer and data propagation through the dedicated registers of the VU. Moreover, a vector operation signal determines the input to the multiplier and consequently the resulting vector operation. Lastly, the VUs simply remain inactive during the IDLE state.

It is possible to allow for multiple components of the same vector to be processed in parallel. However, doing so increases the resources required per VU and thus, depending on the hardware budget, this approach to increasing parallelism may reduce the number of VUs that can be used in the array. This decision involves a tradeoff between vector-level parallelism (process more vectors in parallel) and component-level parallelism (process vector components in parallel). Increasing the component-level parallelism requires the following changes to the VU architecture. First, for each vector component that is to be processed a dedicated subtractor and multiplier is needed. Second, the products produced by each multiplier must be summed up. This can be done sequentially using a cascade of adders, or in parallel using a tree of adders. Notice that the additional adders also increase the hardware utilization per VU. Processing i additional vector components increases the hardware overhead per VU by i additional subtractors and multipliers, and $i - 1$ adders.

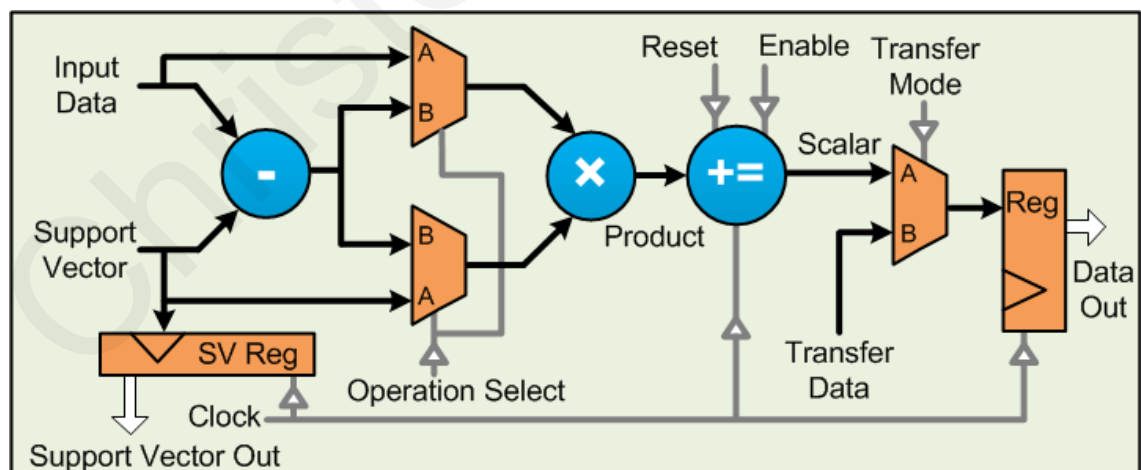


Figure 4-2. Vector unit

B. Scalar Processing Region

The Scalar Units (SUs) are involved in the latter stages of the computation and process the scalar values produced by the VUs, after the vector operations have been completed. Each SU receives the scalar values of the VUs in its row, one per cycle via the right-most VU in the array. The SUs are comprised of two major components, the kernel scalar module (KSM) and a multiply-accumulate unit (MAC). The kernel scalar module performs the scalar operation of each kernel. The MAC unit multiplies each scalar value with its respective alpha coefficient and accumulates the outcome, finally it adds the bias to the accumulated result once the processing of all support vectors is complete. The MAC's bit-width (precision) is determined by the choice of kernel (i.e. the kernel's output bit-width), the number of support vectors (determines the number of accumulations and consequently the precision of the accumulator) and the chosen precision for the alpha coefficients. The architecture of the SU is shown in Figure 4-3.

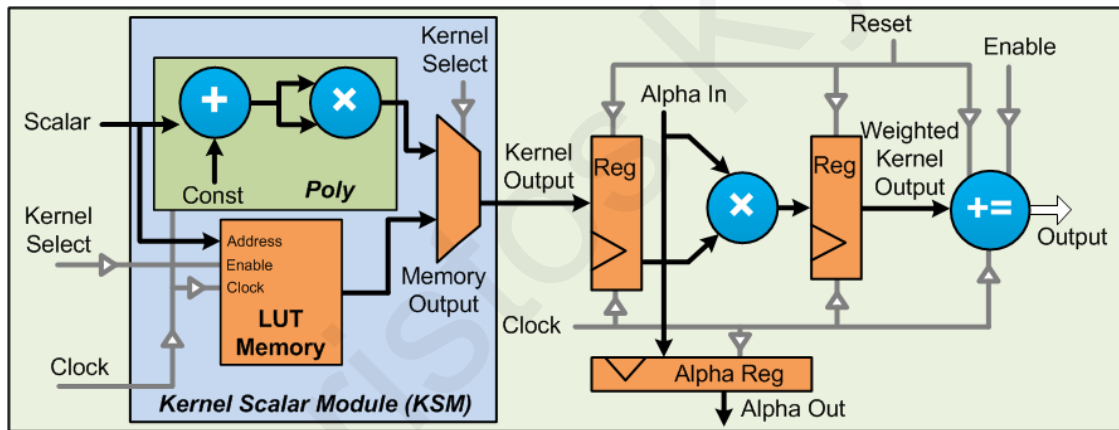


Figure 4-3. Scalar unit

The KSM computes the kernel outcome for each scalar value it receives. The operation it performs depends on the kernel function. The implementation of the kernel function is an important issue. The computations of mathematical functions necessary for the kernel operations may take several cycles if the dedicated circuitry is not present, while the direct hardware implementation of mathematical functions required by kernels, such as division, exponential, hyperbolic tangent and square root is a challenging task and will require many processing resources. Thus, approximation algorithms such as CORDIC and Look-up table

(LUTs) techniques are the most preferred implementation strategy instead of a direct implementation of the kernel function. However, CORDIC-like algorithms require a few cycle to iteratively compute the result using shift and add operations, while on the other hand LUTs only take a cycle to retrieve the result from the memory, and hence, they are the preferred option for the implementation of such functions. The most common function used in literature for image object detection is the 2nd degree polynomial kernel [76],[119]. As such, the necessary components for that specific kernel are hardwired, into a single module called the *poly* module, which are an adder and a multiplier. The LUT approach was used to implement the rest of the kernels. The LUT in the KSM can be initialized to the values corresponding to a specific kernel, let that be an RBF kernel or a polynomial with a degree other than 2. A selection signal (*Kernel Select*) is used to determine whether the *poly* module or the LUT will provide the input to the latter stages. This KSM configuration allows the SU to implement a variety of kernel functions, however, the tradeoff involved is that the precision of the LUT in the KSM impacts the classification accuracy and memory demands. The implementation requirements of the kernel functions result in processing units that may require a lot of resources (high bit-width resources especially), and may also reduce the operating frequency. As such, the SUs are pipelined to reduce long delay paths and increase clock frequency.

One of the key benefits from arranging the processing units in an array structure, in contrast to existing works, is that the resource-hungry components in the SUs are shared amongst multiple VUs and thus are used in computing the scalar value of many support vectors, instead of having dedicated units for each vector operation. This is possible as the vector operations and the produced scalar values have no dependencies between them. Furthermore, due to the systolic nature of the array architecture the scalar values, in the same row, are produced sequentially and since all scalar values will be subject to the same processing operations, it is efficient to process them with the same unit using resource sharing thus, reducing hardware demands and complexity without negatively impacting performance.

C. On-Chip Data Memory Management

Support Vector Machines, like many other machine learning algorithms, exhibit characteristics such as predictable memory access patterns and independent operations, while having to process large amount of training data. Thus, providing parallel access to the training

data is critical in exploiting the inherent parallelism of SVMs. Under these considerations an efficient parallel memory structure was developed that feeds the array of VUs with training data. The memory structure consists of banks of memories (equal to the number of array columns) that supply the array with support vector data through the VUs in the top row of the array (Figure 4-1). The support vectors are distributed amongst the memory banks to allow for parallel access and processing. The memories are arranged in a pipelined structure that facilitates address data movement in the same manner as in the processing array. This helps maintain temporal consistency as well as provide parallel access to the memories since the address data are moved in a pipelined fashion, from memory to memory, avoiding the use of dedicated wires per memory that would have increased the hardware complexity.

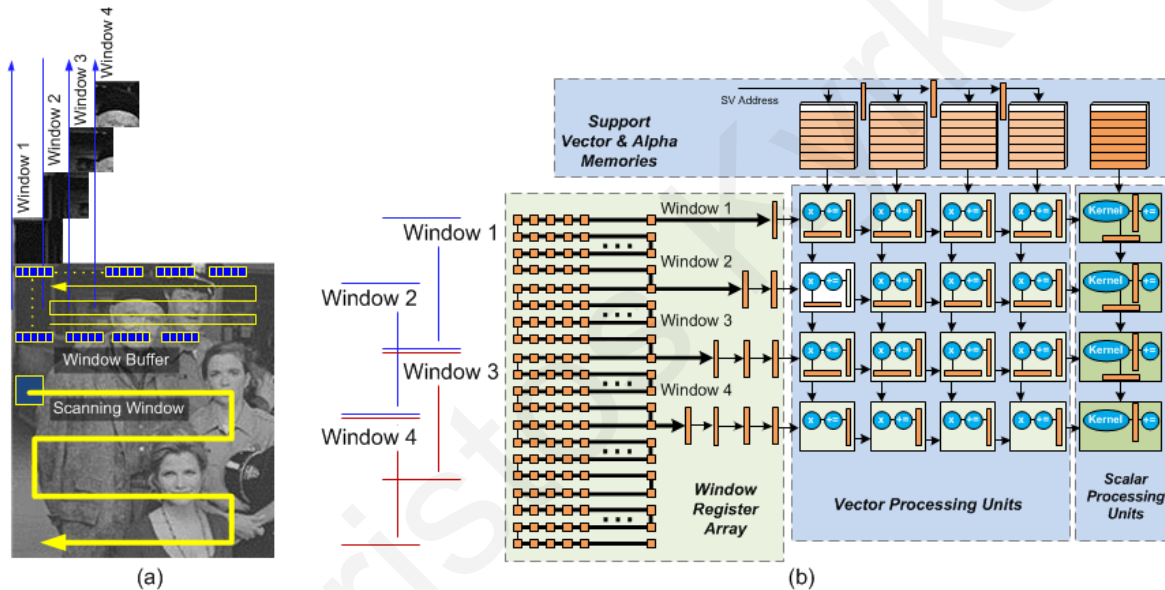


Figure 4-4. Window buffer register array architecture

(a) Part of the image containing search windows is loaded into the register array window buffer (b) Window buffer architecture and interface with the array. The window buffer outputs pixels from multiple windows in parallel.

A window buffer structure is used to provide temporary storage of the input windows and provide parallel input to the array and it is based on a register array structure that is capable of receiving serial input and providing parallel output and is illustrated in Figure 4-4. The register array receives a pixel each cycle until all pixels corresponding to search windows loaded and correctly placed into the array. At that point the output begins and each row of the processing

array receives pixels from a specific register in the register array as shown in Figure 4-4. In this way the overlapping regions contained in more than one window are loaded only once and used for all windows as they are propagated through the register array, thus providing a constant stream of data to the SVM processing array every clock cycle. It is necessary to add delay registers to the outputs of the register array in order to synchronize the pixels with the support vectors and maintain temporal consistency.

4.2.2 Flow of Operation

Processing of the input vectors (search windows) in the array happens in steps. During each step, a number of support vectors, equal to the number of columns in the array, is being processed. The number of steps required is determined by the maximum number of support vectors in each memory unit. For example, if there are 80 columns and memory units, and 120 support vectors, then 40 memory banks will hold 2 support vectors and the other 40 will have 1 support vector each. Processing an input vector will require two steps, in the first step the first 80 support vectors will be processed while the remaining 40 will be processed in the second. At each step, the VUs which do not process any support vectors simply propagate data to the SUs. The arrangement of support vectors in memories is application specific and depends on the available hardware resources as well. If the hardware budget allows it, the array can be made as parallel as possible, otherwise, it can be adapted to the available hardware.

Before the computation of the feed-forward phase begins the array must first be initialized with the SVM parameters. These include the *bias value*, the *support vectors*, the *alpha coefficients*, and the *kernel function data*. The initialization can be done at run time by an I/O controller that interfaces with the array. The memory region must first be initialized with the training data (support vectors and alpha coefficients). The vector operation is selected in each VU via a control signal that is propagated in systolic manner through the array. Another control bit is used to select between the LUT and the *poly* module in the SUs. The LUT must first be initialized with the appropriate data. Initialization is done through the top row SU, which transmits data values through the same pipeline that is used for the *alpha coefficients*.

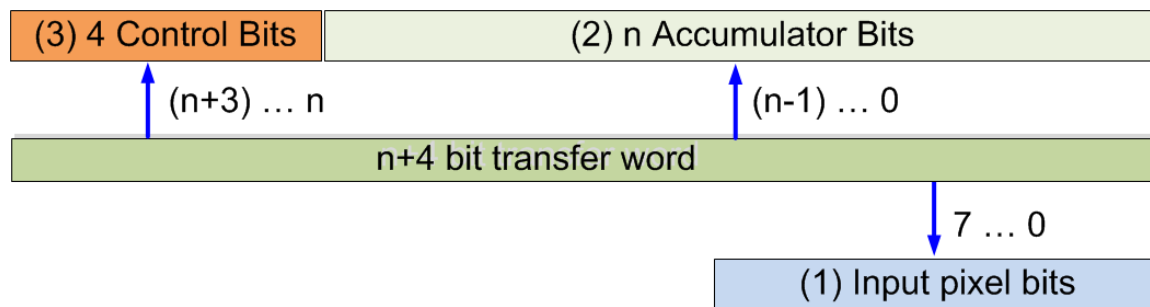


Figure 4-5. Transferred data word in the horizontal direction

Transfer word: (1) In the PROCESSING state the input vector component is transferred in the 8 least significant bits. (2) During the TRANSFERRING state the accumulation result is transferred in the n least significant bits. (3) In both states the four most significant bits are control signals encoded in the following order: *VU reset signal*, *Vector operation select*, *VU enable signal*, *Transfer data select*.

After the array is initialized with the SVM parameters and training data, the classification procedure can be initiated. It begins with the all the array processing elements in the IDLE state. The top-left-most VU is the first one to be enabled after it receives the first components of the input and support vectors, thus entering the PROCESSING state. The neighboring VUs follow next and continue to propagate the vector values and control signals leading more VUs to the PROCESSING state. After $(array_row \times array_column - 1)$ cycles, all the VUs in the array will be in the PROCESSING state, at which point the array will reach its full processing potential. Input vector values and control signals are propagated row-wise, while the incoming support vector values are propagated column-wise by each VU. The transferred control and data values are encoded in a data word as shown in Figure 4-5.

When the scalar value is computed in all VUs, after c (*number of components in vector*) cycles, they all enter the TRANSFERRING state simultaneously, each propagating the computed scalar values towards the right-most VUs, which in turn propagate them on to the SUs. At this point the SUs are enabled and begin processing each scalar value that they receive from the right-most VUs. Along with the SUs, the alpha coefficients memory, which provides the SUs with the alpha coefficients, is also enabled. Each alpha value is transferred through a pipeline downwards to each SU. This is necessary to maintain temporal consistency, as the alpha coefficients must be multiplied with the scalar value of their respective support vectors. The following cycle after the VUs have entered the TRANSFERRING state they are

reset in systolic manner starting from the top-left-most VU, each VU will again enter the PROCESSING state to begin a new vector operation the following cycle after it has been reset. When all scalar values have been processed, the bias is added to the accumulated result to obtain the classification outcome. The transfer from one state to another is facilitated by dedicated control signals that flow in systolic manner through the array avoiding the use of global control signals. From the above analysis the total number of cycles required for the classification of input vectors equal to the number of rows is given by:

$$[m/array_column] \times ((array_row + array_column - 1) + c + reset\ cycle + transfer\ cycle) \quad \text{Equation 4-5}$$

The array processes in parallel as many support vectors as the number of columns in the array. Hence, assuming m support vectors in a training set, $[m/array_columns]$ repetitions are necessary to process the input vector with all support vectors.

4.2.3 Scalability and Implementation Issues

An array consisting of i columns and j rows, will have $i \times j$ VUs, j SUs, and it can process j parallel input vectors and i parallel support vectors. The hardware resources are determined by the number of SUs and VUs, as well as minor overheads for wiring and control of the array. Increasing the number of rows requires additional VUs equal to the number of columns, and a single SU. On the other hand, increasing the number of columns requires additional VUs equal to the number of rows in the array. Increasing the array size in either way increases parallelism and the hardware overheads to the array rows and columns increase linearly as well. The array rows are however, constrained by the memory I/O bandwidth and the array columns are constrained by the number of support vectors and support vector memory bandwidth and capacity.

4.2.4 Multiclass Classification Support

The proposed array architecture is not only suitable for binary SVM classification problems, but can also be extended to handle multi-class classification as well, with minimal hardware overhead. An example of a multi-class classification problem is face recognition,

where the input window must be classified in one of the possible candidates in a face database [137]. To handle such problems the rows in the array must be decoupled so that they can work independently towards different classification problems. Each row must be supplied with its own set of support vectors and alpha coefficients. The following modifications must take place to allow the architecture to handle multi-class classification problems (also shown in Figure 4-6).

- i. Each VU requires a multiplexer to select between the support vector from its above VU, or a support vector memory unit.
- ii. Multiple controllers are required, one per row. Each controlling the operation of a row under different classification parameters. However, only one is necessary when the system operates as an array.
- iii. The left-most VUs will also require multiplexers to select between different control signals. When operating as an array the control signals will come from one central control unit, while when each row operates independently the control signals will come from the dedicated control units
- iv. Each SU will also require a multiplexer to select between the alpha coefficients from the previous SU when operating as an array, or from one of the alpha coefficient memories.

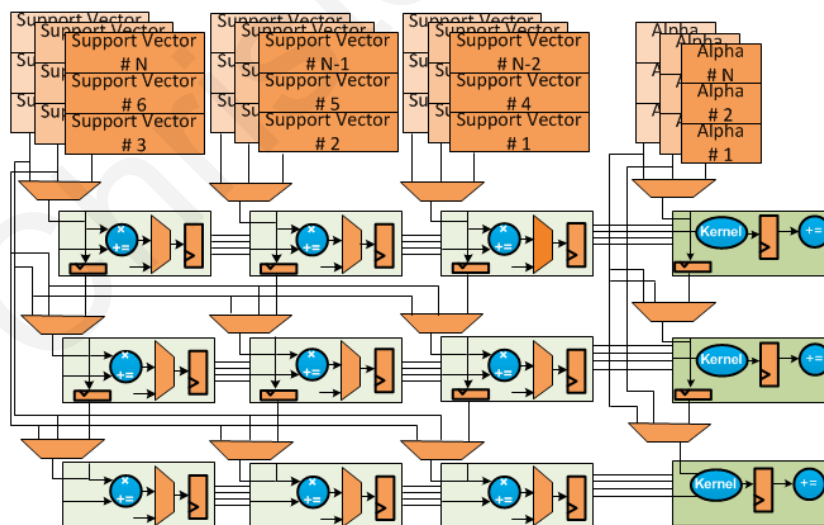


Figure 4-6. Support vector machine processing array multiclass support

The main advantage that stems from enhancing the array with the additional multiplexers is that the array can operate in various configurations depending on the application demands. First, it can operate as a fully parallel systolic array to speed up a single object detection problem, using one of the available training set memories. Second, each row can work independently on one classification problem (face recognition). Finally, any combination of the above two configurations is possible, such as multiple systolic arrays, or a single systolic array with multiple independent rows. This flexibility allows the enhanced array processing engine to adapt to a variety of object detection scenarios and specific application demands, making it suitable for embedded environments which exhibit a high degree of variability.

4.2.5 Experimental Methodology and Evaluation Results

A prototype of the array architecture was implemented on an FPGA platform as a proof of concept, and evaluated it using three popular detection applications. Prior to detailing the FPGA implementation, the factors that affect the performance of an object detection system are first discussed, followed by details on the training methodology for the three applications.

A. Performance and Constraints

There are several factors that impact the performance of an object detection system, but first, it is important to consider the metrics used to measure performance. An image object detection system is characterized by how accurately it can classify data as well as how many image frames it can process per second. Thus, the two commonly used performance metrics are the *detection accuracy*, and *frames per second* (FPS) or *frame rate*. A minimum performance of 30 FPS is required in order for an object detection system to be capable for *real-time* video processing. However, certain applications that process multiple streams may require higher processing rates.

A factor that affects both the detection accuracy as well as the frame rate is the number of support vectors. The fewer the support vectors the better the performance, since less operations are required per input vector. However, the detection accuracy may be reduced when fewer support vectors are used. Also, the detection accuracy is also affected by the bit-width used to represent the training data in hardware. However, if the bit-width is chosen

appropriately with respect to the targeted application domain, the resulting accuracy loss can be minimal.

Performance is also affected by several other factors, typically encountered in object detection applications. The first is the number of search windows that need to be processed per frame. This is determined both by the size of the object of interest, and the search window overlap. When the object of interest has a relatively small size, and consequently a small search window is used, more windows will be generated per input frame increasing the processing time per frame. Conversely, if the targeted object of interest is large, fewer windows will be generated. The window overlap between successive windows also determines the number of generated windows, and is determined by the size of the object of interest. Choosing the appropriate window overlap involves a tradeoff between the granularity of the window search and as such the detection accuracy, and the resulting FPS.

The input image size is also equally important to the performance of an object detection system. A larger input image will generate more search windows increasing the time needed to process the whole frame. At the same time the number of downscaled versions of the input image will increase to account for objects of different sizes. The input image size and number of downscaled frames have a greater impact on performance when the search window size is small and as a result a large number of windows will be generated.

Another factor that limits the performance of object detection systems is the memory access mechanism and I/O capabilities. Memory access for both the input and support vectors is of great importance as it limits the capabilities of a system for parallel processing. It is important to have parallel access to the training set and at the same time fetch multiple search windows, in order to take full advantage of the capabilities of a fully parallel architecture. Additionally, it is important to consider where the data will be located, off-chip or on-chip. Input data are usually stored off-chip as they arrive from an external image/video acquisition source. The training data on the other hand, can be stored on-chip if the memory is available, otherwise, they are also stored off-chip as well. The latter may decrease performance if the number of support vectors is large, as off-chip communication will become the bottleneck to the system performance.

Finally, the operating frequency of the object detection system greatly impacts the performance. For FPGA based designs this is a limiting factor as fixed routing and LUT placement may not allow for a design to operate at its full potential. High frequencies can be achieved by regular and modular designs with small critical paths, such as the proposed array-based hardware architecture.

B. SVM Training Set and Parameters

Three object detection applications are used as benchmarks for evaluating the proposed architecture: pedestrian detection [80],[95], car side view detection [97], and face detection [76],[136]. All three detection problems are interesting for the proposed implementation, as they can be applied in intelligent embedded environments for surveillance and security purposes, as well as traffic and street monitoring. Furthermore, the three detection problems concern different objects, consequently, each one has different detection parameters such as search window size, window overlap, and number of downscaled images. As a result, the architecture is evaluated under different application parameters and is analyzed for its suitability for generic object detection. Details of the parameters and characteristics of each application are shown in Table 4-2. All detection problems concern grayscale images corresponding to 8 bit pixel values. Training of the SVM models was done using the SMO algorithm implementation [138] provided in MATLAB R2010b [46], with publicly available data sets from [139], [140], [141], each consisting of training and test sets for both the negative and positive classes. To further test and evaluate the generalization capabilities of the trained SVM models full test frames obtained from [142], [140], and [143] for face, car side view, and pedestrian detection respectively were used. These frames were rescaled to 320×240 pixel images and used to evaluate the performance of the hardware implementation in terms of detection accuracy and frame rate. The most generally used 2nd degree polynomial and RBF kernels were used in the experiments. The former was found to be efficient for object detection applications[119], while the latter is also widely used in many applications with very good detection results [37]. The best SVM model was selected based on the following two criteria. First, the memory required to store the support vectors must not exceed the available memory of the experimental platform. Second, the selected model must maintain good accuracy rate on the full frame images for each application. Using selected full-frame

test images from the datasets [140],[142], and [143], the accuracy of each SVM model was evaluated (for each application the number of windows that must be processed per frame is shown on Table 4-2).

The true positive (TP) and false positive (FP) rates are given in Table 4-2. True positives correspond to rectangles on the resulting output image which correctly contained an object of interest, all other image regions that are marked as containing an object of interest, but do not in fact contain such an object, are categorized as false positives. The reported accuracy rates of course depend on the training data, and obviously a stronger training set can increase the detection accuracies. The SVM model parameters selected for each detection problem are also given in Table 4-2.

TABLE 4-2: SUPPORT VECTOR MACHINE PROCESSING ARRAY PARAMETERS AND RESULTS

Applications	Window Size/ Dimensionality	Downscaled Images	Window Overlap	Total Search Windows	Support Vectors	Kernel Function	Accuracy		FPS
							TP	FP	
Face Detection	19×19 (361)	5	5	4405	400	Polynomial degree=2, const=0	77%	0,2%	~40
Pedestrian Detection	18×36 (648)	4	8	1955	467	RBF sigma=5	76%	0,35%	~46
Car Side-view Detection	100×40 (4000)	4	10	790	74	Polynomial degree=2, const=0	78%	0,2%	~122

C. FPGA Implementation and Evaluation

A prototype of the array processing architecture was implemented to perform the SVM feed forward phase for the three applications targeting the ML505 evaluation platform [88]. The selected platform is equipped with a Virtex 5 LX110T FPGA, an external DDR2 DRAM with a capacity of 256MB, a DVI output, a compact flash card reader, and 64 DSP units (embedded multipliers and accumulators), which makes it suitable for evaluating object detection algorithms that require large amounts of memory as well as visual verification of results. A prototype of the presented array processing architecture was developed on the FPGA, which interfaced with an embedded Xilinx Microblaze soft-processor [144] for I/O purposes. The overall system is illustrated in Figure 4-7, while resource utilization, obtained using Xilinx

ISE 12.4 and EDK 12.4, for both the Microblaze system and the implemented array are given in Table 4-3. Performance results (frame rate and detection accuracy) are shown in Table 4-2. Finally, the FPGA prototype running face detection is illustrated in Figure 4-8-(a) and some examples of test image detection results are shown in Figure 4-8-(b).

TABLE 4-3: ARRAY PROCESSING ENGINE FPGA SYNTHESIS RESULTS

FPGA Resources	Logic Elements		Embedded Multipliers DSP48E (64)	Block RAMs (148)	Frequency (MHz)
	Slice LUTs (69120)	Slice Registers (69120)			
Processing Array	57296 (82%)	23220 (33%)	40 (62%)	83 (56%)	100
Microblaze I/O System	6221 (9%)	7251 (10%)	3 (4%)	37 (27%)	
Vector Unit	153 (<1%)	50 (<1%)	---	---	
Scalar Unit	288 (<1%)	372 (<1%)	10 (15%)	---	

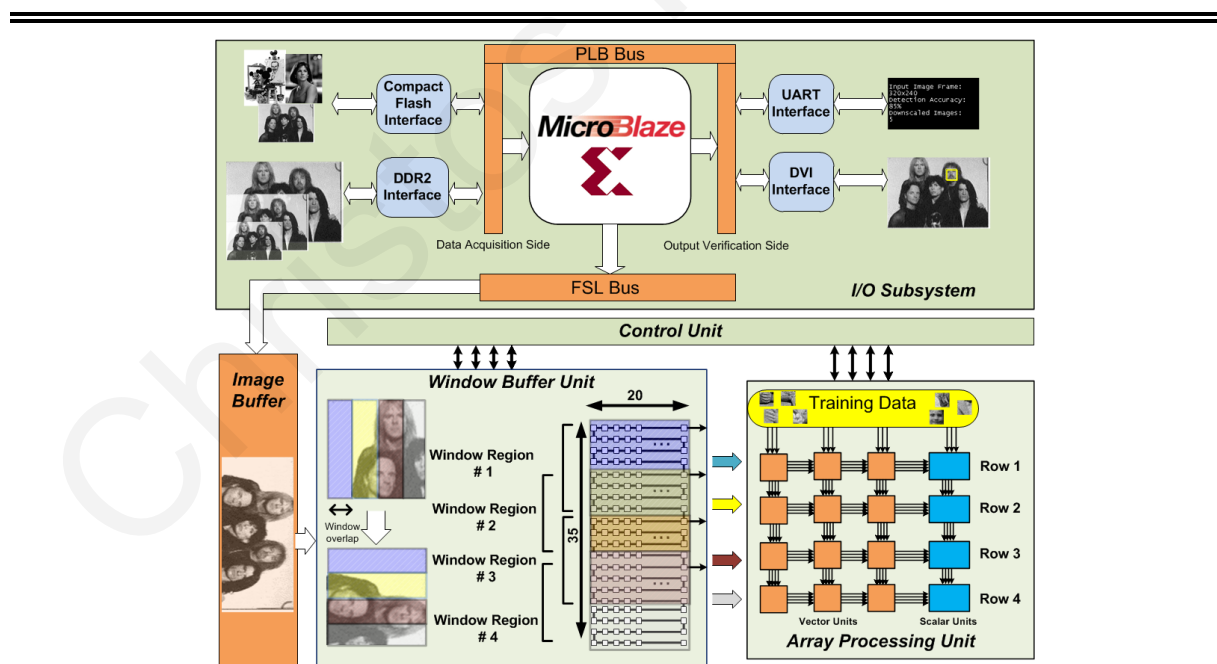


Figure 4-7. Implemented FPGA support vector machine array processing system

TABLE 4-4: SUPPORT VECTOR MACHINE ARRAY HARDWARE PARAMETERS

Parameters	VU Multiplier	VU Accumulator	Alpha Coefficients Memory	KSM LUT Memory
Bitwidth/ Memory	8×8 bits	28 bits	3 KB	2 KB
Parameters	KSM Polynomial multiplier	Alpha Coefficients	Alpha Coefficient Multiplier	SU Accumulator
Bitwidth/ Memory	28×28 bits	24 bits	56×24 bits	90 bits



(a)



(b)

Figure 4-8. FPGA system prototype

(a) SVM Array Processing Prototype on a Virtex 5 FPGA running face detection (b) Detection Results

Microblaze handles tasks such as system supervision, control and data transferring to and from the array, while the array is responsible for the SVM classification. Microblaze communicates with external components via dedicated interfaces for data transfer and monitoring purposes. Input image frames were initially stored in a Compact Flash card (acting

as the image acquisition source) and loaded to the external DDR2 DRAM prior to the detection phase. Microblaze retrieves pixel data from the external DRAM and sends it to the array via the Fast-Simplex-Link (FSL) bus interface. Evaluation and verification as well as overall system monitoring were done using a serial communication interface, and a DVI interface to output the image frames with the detected objects on a monitor.

Object detection applications exhibit a high degree of data reuse, as a large amount of the currently processed window will also be used for the next window. To reduce the external I/O memory accesses a suitable memory hierarchy was developed and integrated into the architecture. At the first level a memory block is utilized to store an image region (the whole image if possible), while at the second level a register array window buffer is used to store the active image region (i.e. the search windows that are currently processed by the array) and feed the array with data. The window buffer has a capacity to store 4 windows and has a size of 20×35 pixels. A 4-row by 80-column array was implemented based on the proposed array architecture and synthesized targeting an FPGA prototype. A total of 320 VUs, 80 memory units and 4 SUs were implemented in the prototype. Each memory unit was allocated a capacity of 4 KB, thus, a total of 320 KB of FPGA block ram was allocated for the training set. With the rest of the memory available on the FPGA whole input image can be stored on chip to reduce off-chip memory accesses.

The VUs were mapped on the FPGA custom logic as they did not consume many FPGA resources. The SUs, on the other hand, were more demanding (because of the alpha coefficient multiplier which is the critical path of the design) and thus were mapped on the dedicated DSP units of the FPGA. The entire SU was pipelined in an effort to maintain high frequency. It must be noted that the number of DSPs on the FPGA does not limit the number of SUs that can be instantiated, as any additional SU can be instantiated on the FPGA custom logic. Table 4-4 summarizes the hardware parameters of the implemented array. The clock frequency of the design was set at 100 MHz, which is the system clock frequency of the FPGA. Higher clock frequencies can be achieved by further optimizing the design, especially the SU that is the system bottleneck in terms of the operating frequency. However, for prototyping purposes no further optimizations were carried out regarding frequency.

Depending on the application I/O demands and interface, the structure can be chosen accordingly to provide the necessary tradeoff between performance and hardware area. The memory units were initialized according to each application's training data. If the number of support vectors necessary for each application (as derived by the training set) is larger than the columns of the array, the computation will have to be repeated (using time-division multiplexing) until all support vectors are processed for each input vector.

D. Performance Results and Discussion

Typically performance in object detection systems is measured in frames per second (FPS). The processing of a frame also includes processing all downscaled versions. The time needed for the proposed architecture to process a single frame can be calculated using (Equation 4-5) from Section 4.2.2, and it depends on the total number of windows that must be processed for all scaled image versions, the number of windows that are processed in parallel, the input rate, and the operating frequency of the array. Also the structure of the array (number of units and size) is important to the resulting performance. To evaluate the performance of the architecture on the FPGA images of 320×240 pixels are considered for the three benchmark applications, selected from publicly available data sets from [142], [140], and [143]. These images contain a varying number of objects in various sizes. The number of generated windows per image frame (including the downscaled versions) for each application was then computed (given in Table 4-2).

The implemented array prototype can process four windows in parallel, while the operating frequency of the FPGA is 100 MHz, and the input rate to the array is four pixels every cycle from the register array. Using this information, the resulting frame rates achieved by the proposed architecture are 40, 46 and 122, for face detection, pedestrian detection and car-side-view detection respectively, which are sufficient for real time object detection. It must be noted, that the frame rate of the proposed architecture depends only on the number of search windows, and architecture-specific details (such as number of units). As such, computing the performance is not affected by variable parameters such as number of objects or their size, since all search windows will go through the whole classification procedure. The resulting frame rates suggest that the system is capable of processing larger input images. The performance of the proposed architecture depends linearly on the image size (i.e. number of

windows in the image). As such, when the input image size increases above a certain size it is expected that the frame rate will decrease below the adequate real-time performance levels (30 FPS) since the amount of data that needs to be processed (number of windows and downsampled image versions) increases as well. This is also true for most hardware implementations found in the literature. To handle higher resolution images or an increasing number of search windows it is possible to use cascade SVMs [54] to speed up the classification procedure of a single window. However, a hardware architecture for this approach will be the main focus of the following section.

A single array structure was used to evaluate the performance of all three benchmark applications to illustrate that a single structure can be used in a variety of applications. As such the bit-widths of the processing units were chosen to cover the most demanding application. However, if only a single application was to be considered, the array could be optimized for that application in terms of bit-width for each processing unit, thus permitting for more units to be implemented, leading to higher frame rates. The car side view detection application has the highest frame rate, an order of magnitude higher than the other two applications. This is primarily due to the fact that it must process an order of magnitude less support vectors and generated windows compared to the other two applications. This shows the impact of the search window size with respect to the input image size, the primary reason for the small number of generated windows. On the other hand, the face detection application produces the largest number of generated windows, when compared to the other two applications, as it has the smallest search window size, resulting in the lowest frame rate. The pedestrian detection frame rate is higher than that of face detection since it has four times less generated windows to process, even though it requires processing more support vectors. In addition the frame rate suffers for both the pedestrian and face detection applications because the number of columns is less than the number of support vectors for each application. As a result the input vectors must pass through the array multiple times until they are computed across all support vectors. Overall, for a variety of window sizes and amount of training data, the targeted architecture under the limitation of the given FPGA resources, manages to offer adequate real-time performance for all three benchmark applications.

Detection accuracy is also an important performance measure in the context of object detection. The accuracy when dealing with SVMs is determined by the support vectors and alpha coefficients that are derived during training, and their representation in hardware. Through software simulations the appropriate bit-width representation was determined so *no accuracy loss* was observed for the test set of each application. The sufficient number of bits to satisfy all three benchmark applications was found to be twenty-four. By using this bit-width representation the hardware implementation of the proposed architecture maintains the same accuracy rates as the equivalent software SVM models in MATLAB. These accuracies are similar to software implementations found in the literature [97], [76], [20].

E. Comparison with Other Works

Comparing architectures evaluated with different applications other than video object detection is not practical, as factors that possibly affect performance (input image size, number of downscaled images) were not considered in related works. Consequently, a comparison is made based on the provided information. Related works that have proposed the hardware implementation of the SVM feed-forward phase include [108], [115], [92], [118], and [119]. From these works, [115], [92], and [118] propose architectures which depend on the vector dimensionality (i.e. target a specific application) and as such would not be applicable for different object detection applications. The vector coprocessor implementation in [108] primarily targets SVM training, and thus no clear results are given in terms of classification performance, which the authors measure in GMACS. The presented coprocessor consists of 100 vector processing elements clocked at 115 MHz providing a sustained performance of about 10 GMACS. Using a similar configuration, with 100 VUs clocked at 100 MHz, the presented array architecture can also provide 10 GMACS. However, given that in the presented implementation three times more VUs were used, the resulting compute performance is much greater. In [119], matrix bar-code detection in images is performed using a search window size of 16×16 pixels (256 vector components), and 88 support vectors. The implemented vector coprocessor aims at parallelizing the processing of a single vector and as such it processes input vectors sequentially, and each requires 352 cycles to be processed. By utilizing a similar configuration to the one used for the benchmark applications (4 rows and 80 columns), the presented architecture can process 4 input vectors in 443 cycles.

In both cases the presented architecture shows that due to its parallel systolic nature and modular design, it can outperform existing works.

4.2.6 Discussion and Impact

An array processing engine for object detection with monolithic SVMs was presented in this section that can achieve real-time performance (40-122 FPS for three benchmark applications) while maintaining high detection accuracies (76-78% for a variety of applications). The architecture scales linearly to the hardware budget taking full advantage of its modular and regular design, while providing true parallel processing for both input and support vectors. It is also capable to implement different kernels through the generic vector and scalar processing units. Also the demanding kernel implementation has also been addressed by sharing it amongst many vector processing units. Furthermore, the same array structure can be used for different applications regardless of the window size, number of support vectors and image size. Additionally, using the enhanced version of the proposed architecture it can be configured to operate in a variety of modes and is able to adapt to different application demands (such as multi-class applications). Overall the proposed architecture is capable of real-time SVM-based object detection while providing a configurable detection platform that can operate in a variety of embedded object detection scenarios, and adapt to specific application and designer demands. To enable higher-frame rates as the image size increases or the number of search windows increases it is possible to use a cascade of SVMs similar to the Viola and Jones attentional cascade discussed in the previous chapter.

4.3 Hardware Acceleration of Cascade SVMs

Cascade Support Vector Machine (SVM) classifiers are widely used for various embedded applications, such as object detection, and provide speedups over monolithic (single) SVM classifiers. However, multiple SVM classifiers need to be implemented in this approach making it challenging to achieve real-time classification with low resource utilization and power consumption, all key constraints for embedded systems. Existing SVM hardware architectures implemented on FPGAs consider only monolithic SVM classifiers. Hence, such

hardware architectures are not optimized for problems where the majority of data belong to one of the two classes. As such, designing specialized hardware architectures for multistage cascade SVMs based on existing approaches is a challenging task due to the increase in the number of classifiers, especially for embedded applications which require low power, real-time operation, and often with limited available resources. In an attempt to improve the suitability of cascade SVMs for embedded applications a suitable hardware-efficient architecture tailored to the cascade processing flow is presented in this section. Specifically, it is a hybrid hardware architecture of sequential and parallel processing modules that exploits the cascade SVM flow, where classifiers at the beginning are used more frequently than subsequent stages, to efficiently utilize the available hardware resources while providing real-time classification. In addition, a method to reduce the hardware complexity of cascade SVMs is presented, which relies on rounding off the SVM training data of the early cascade stages to the nearest power of two values to replace multiplications with shifts. Thus achieving a reduction in the resources needed for the architecture implementation. Finally, a novel approach to reduce the samples that reach the more computationally demanding latter cascade stages is explored, by evaluating the responses of the early cascade stages, leading to improved classification speeds.

4.3.1 Challenges in the Acceleration of Cascade Support Vector Machines

It is possible to speed-up SVM-based classification for a certain class of applications, such as object detection, that exhibit the following characteristics: (a) the majority of the samples presented to the classifier belong to the negative class and (b) the majority of negative samples can be easily distinguished from positive samples. To this end, works in literature have tried to take advantage of these two observations by utilizing stages of SVMs of increasing complexity, which are sequentially applied to the input data (Figure 4-9). Stages of SVM classifiers that mostly follow a cascade structure [54],[55],[56], and in other cases a tree structure [59], [145], [91], sequential evaluation of SVs, [58] have been implemented in software. These works, shown in Table 4-5, demonstrated speedups as well as improved accuracy over monolithic SVM classifiers. The stages at the beginning of the hierarchy have lower computational complexity (i.e. need to process only a small number of SVs) and are

TABLE 4-5: SUPPORT VECTOR MACHINE CASCADE SYSTEMS OVERVIEW

Work		Kukenys [55]	Heisele [54]	Romdhani [57]	Ma [56]	Presented Work
Application		Human Eye Detection	Face Detection	Face Detection	Face Detection	Face Detection
SVM Cascade Structure	Stages	N/P	5	7	6	4
	SVM Type	RBF Kernel	4 Linear stages & 1 2 nd degree Polynomial stage	RBF Kernel	5 linear stages & 1 2 nd degree Polynomial stage	2 linear stages & 2 Polynomial stages
	Number of SVs	100 Reduced Set Vectors from 1796 SVs	N/P	100 Reduced Set Vectors from 8291 SVs	N/P	2 for linear SVMs, 20 for first Polynomial, 100 for second polynomial
Additional Methods, Features, Preprocessing		N/P	Multi-resolution window processing, Feature Reduction, Histogram Equalization	N/P	Histogram Equalization & Lighting Correction	Local Binary Pattern Histogram Descriptors
Training Set	Database	N/P	[139] with additional samples	N/P	N/P	[139] with additional samples
	Positive Samples	1066	9662 for linear 2429 for polynomial	3600	5000	~6000
	Negative Samples	2132	33045 for linear 4548 for polynomial	25000 (additional 110000 samples using bootstrapping)	Collected from 63 images	~50000 Collected from various negative images
Classifier Window Size / Vector Dimensionality		20×20 (400 vector dimension)	3×3,4×4,11×11, 19×19 for linear 19×19 for polynomial	20×20 (400 vector dimension)	20×20 (400 vector dimension)	20×20 (400 & 1062 vector dimension)
Scale Factor		N/P	5 scales	1.42	N/P	1.2 (18 Scales)
Test Databases		N/P	[142]	N/P	[142]	[142],[146],[147]
Test Image Size		500×300	320×240	N/P	N/P	800×600
Detection Speed		0.389 seconds per image	4 frames per second	N/P	N/P	40 FPS
Detection Accuracy		88% CD	80% CD	TP: 80.7% FP: 0.001%	80.6% CD	TP: 80% FP: 0.001%

¹ These figures are not average but rather examples stated in the work

N/P – Not Provided | N/A – Not Applicable | CD - Correct Detection | TP - True Positive | FP - False Positive

tasked with removing the majority of samples from the negative class. The latter stages then are able to perform more accurate classification on the remaining samples, which, however, incurs a higher computational cost (i.e. need to process more SVs). Hence, an input sample needs to pass all stages to be classified as positive (Figure 4-9). Under this scheme a large amount of input samples are discarded early in the classification process by the stages at the beginning of the cascade, resulting in significant speedups. In addition, it is also possible to use the reduced-set-method [51], to reduce the number of support vectors required by the non-linear kernel stages in order to further improve classification times. Furthermore, since the latter stages need to better discriminate between positive and negative samples, feature extraction algorithms may be used to improve accuracy, which however, further increases computational demands.

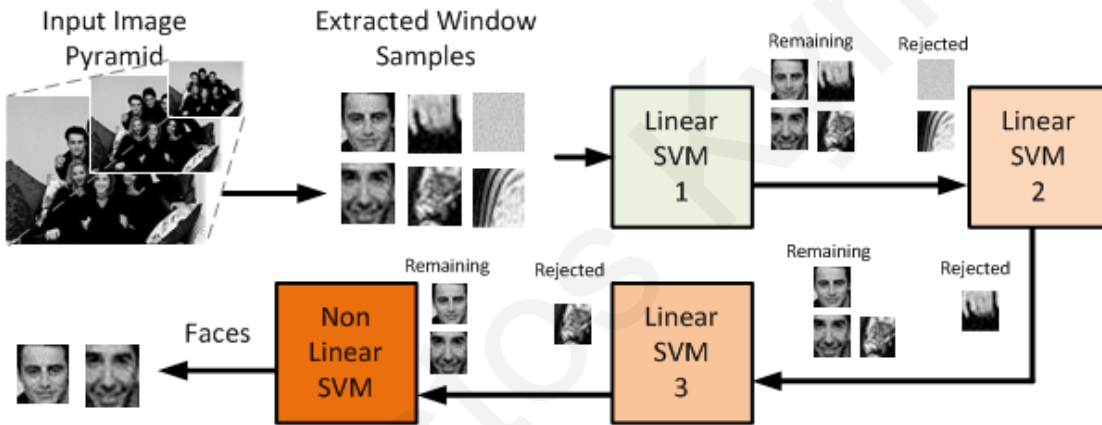


Figure 4-9. SVM cascade classification process overview

4.3.2 Hybrid Hardware Architecture and Optimization Approaches

Cascade classifiers have demonstrated significant speeds over single SVMs, however, it is still challenging to achieve real-time performance, especially as the amount of data that needs to be classified increases. Hence, a parallel hardware architecture is presented aiming at providing higher classification throughput and a hardware reduction method leading to a more compact hardware implementation suitable for embedded system applications. In addition, the following sections also outline a novel way to improve classification speed by taking advantage of cascade classification information to reduce the amount of data samples that

reach the more computationally-intensive latter cascade stages. Finally, in many classification problems some form of feature extraction/preprocessing method needs to take place in order to deal with different invariances and improve detection accuracy. Since cascade classifiers have been prominently used for object detection one such popular method used in a variety of object detection applications namely local binary pattern (LBP) descriptors [62], is also integrated to the architecture.

A. Cascade SVM Hardware Reduction Method

Existing cascade SVM classification schemes utilize a hierarchy of SVM classifiers which can be different classifiers or expansions of a single classifier. Nonetheless, the common feature of these cascade structures is that stages at the beginning of the cascade usually require processing less SVs than subsequent stages making them computationally less demanding. This is because the objective of the early SVM stages is to guarantee that the positive samples will go through to the final stage while a large amount of negative samples will be discarded rather quickly. However, this implies that a few negative samples will be classified as positive. In contrast, subsequent stages need to be more accurate and discriminate better between positive and negative samples, and hence build decision functions with more SVs.

The fact that the early SVMs stages are not optimal classifiers can be exploited to reduce the resources required for their hardware implementation by adapting their parameters (SVs and alpha coefficients), while maintaining their ability to discard a large amount of negative samples. The proposed hardware reduction method is to approximate the support vector and alpha values of the low complexity kernels with the nearest power of two values. This will result in all the multiplication operations in the SVM classification phase (the kernel dot-product calculations and computations related to the alpha coefficients) becoming shift operations. Additionally, since the support vectors and alpha coefficients are now power of two values there is no need to store the binary representations of decimal numbers but only shift data (shift amount, shift direction, and number sign). Hence, this results in an *adapted cascade SVM* with reduced storage and computational demands. However, by approximating the support vectors and alpha coefficients the resulting classification accuracy will be different from that of the *initial SVM cascade*. The receiver-operating-characteristic (ROC) curve of each cascade stage rounded off to the nearest power of two is used to adjust its accuracy, to

similar rates of that of the initial cascade stages. The ROC curve shows the performance of a binary classifier by illustrating the corresponding true positive and false positive rates, given a test set. As such, by setting the appropriate threshold the performance of the adapted stages in the SVM cascade can be adjusted to match the true positive rate of the initial SVM cascade stages. This is necessary in order to maintain the true positive rate. There are trade-offs which stem from changing the original classification model. Specifically, the reduced computational and storage requirements come at a cost of an increase in the false positive rate of the adapted classifiers. However, the overall accuracy tends to meet the accuracy of the final classification stage and hence the increase is not significant. Adapted stages, which do not yield the targeted accuracy, are reverted back to the initial model. The process is summarized in Figure 4-10.

The hardware reduction process takes place after the cascade structure is decided, meaning that the kernel function, and number of support vectors or reduced-set-vectors (RSVs) for each SVM cascade stage are determined. As such, the proposed method can easily be used with different SVM training frameworks. Furthermore, the method does not depend on the specific hardware architecture used for the implementation of the cascade and as such can be optimized to different architecture requirements.

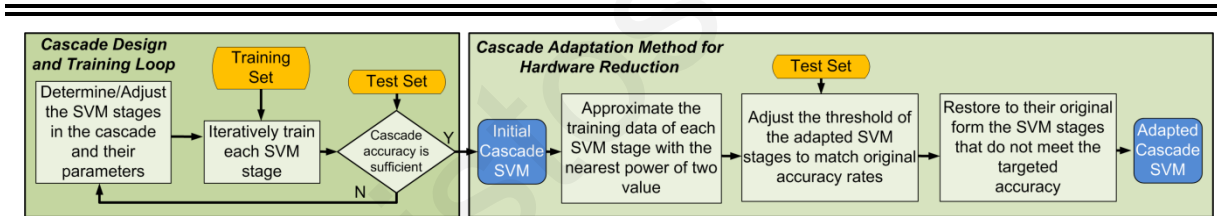


Figure 4-10. Hardware reduction method

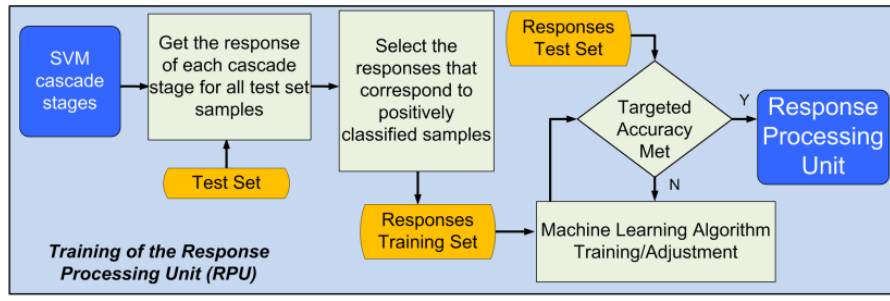
The initial cascade support vector machine is adapted to reduce its hardware processing requirements after the cascade training phase

B. Cascade Response Evaluation Method

Exploiting cascade information of the individual classifier stages was usually done in the training phase to eliminate samples from the training set so that the training time could be reduced. Only few attempts have been made to do something similar in the classification phase with reported methods [148], [149] performing some sort of joint operation (AND-OR) on the outcome of the cascade stages in order to correct/reevaluate the detection result. Such

methods are usually used to improve detection and require that all the stages process the input data in order to reach a decision. However, this means that the overall detection speed is reduced. In order to improve performance there is a need for a mechanism that can indicate whether an input sample needs to move on to the more computationally demanding stages. A way that this can be done is by examining the responses of early cascade stages in order to rapidly eliminate data samples prior to reaching the latter stages. It is based on the observation that when looked at collectively, the responses of the individual cascade stages can exhibit patterns which can help in discriminating between samples belonging to different classes. This adds an additional dimension to the cascade classification phase that amongst others can be used to speedup the overall process.

Such a response processing mechanism can be constructed as shown in Figure 4-11. First, test samples must be classified by the selected cascade stages so that a response feature vector can be constructed. Next, the samples which are predicted as positive class, i.e. have passed all stages, are chosen so that their corresponding response vectors form a new training set of negative and positive response feature vectors. Using this new training set a machine learning algorithm, which will act as a response evaluator, can be used in order to discriminate between different responses. This process is further outlined in Figure 4-11. Of course, the positive and negative samples can often have similar cascade responses. Hence, the training goal for the machine learning algorithm is to make sure that the positive responses will be correctly classified so that the true positive accuracy of the whole cascade is not affected. With regards to responses corresponding to negative classes, any correct classification is beneficial since those samples will not need to be classified by the final stage. The desired true positive rate can be adjusted experimentally by setting an appropriate threshold value. This is a general approach of handling the cascade responses and thus can be used similarly to benefit both software and hardware implementations. With regards to software implementations the additional computations necessary for the latter cascade stages are eliminated, while for hardware implementations, the reduced workload can result in more compact architecture implementations for the latter stages.



1. {Input: Set of positive and negative samples not used in cascade SVM training $(x_1, y_1) \dots (x_n, y_n)$ where $y_i = \{-1, 1\}$. Cascade SVM stages.}
2. **Initialization:** Select the desired first m SVM cascade stages
3. **For** $i = 1, \dots, n$
 - For** $j = 1, \dots, m$
 - a. Classify sample x_i with $SVM_stage(j)$.
 - b. Get $SVM_stage(j)$ response $\rightarrow SVM_out(i, j)$.
 - c. If predicted "negative" move to the next sample.
 - d. If predicted "positive" move to next stage.
4. Collect responses of the samples that have been predicted as "positive" by all m stages $\rightarrow [SVM_out(i, 1 \dots j) y_i]$.
5. Separate the data set into training D_{Tr} and test sets D_{Te} .
6. Train a machine learning algorithm using D_{Tr} .
7. Evaluate trained model accuracy using D_{Te} .
8. Adjust threshold to achieve the desired true positive accuracy rate using ROC curve.
9. {Output: Response Evaluation Algorithm}

Figure 4-11. Cascade response evaluation method

(Top) A machine learning algorithm is trained to perform response evaluation of the early cascade stages to further reject data instances to the final stage. (Bottom) Response evaluation method steps.

C. Cascade Hardware Architecture

Due to the nature of the cascade classification scheme each SVM stage will have fewer input data to process and more SVs to process than the previous. Hence, efficient hardware architectures need to take into consideration the throughput and processing needs of each stage in the cascade. Accordingly then, the proposed hardware architecture (Figure 4-12) for the cascaded SVM classifier consists of two main processing modules, which provide different parallelism with respect to the input data and SVs, in order to meet the different demands of the cascade stages.

The first is a fully parallel processing module (PPM) which performs the processing necessary for all the adapted SVM stages. The second is a sequential processing module (SPM) which is optimized for the high complexity SVM stages which demand processing a large number of SVs but only a fraction of the input data. Thus parallelism focuses on processing more SVs in parallel. In addition, a shift register structure is used to provide

sequential and parallel data access to the two processing modules, and also to take advantage of potential data overlap and reduce memory I/O. A frame buffer is employed to hold part of the image for fast local access. The cascade response information processing is implemented with a low-latency, low-resource consuming neural network architecture to minimize hardware overheads while boosting performance. Finally, the architecture incorporates a specialized processor that performs local binary pattern (LBP) histogram extraction which is used as features for classification.

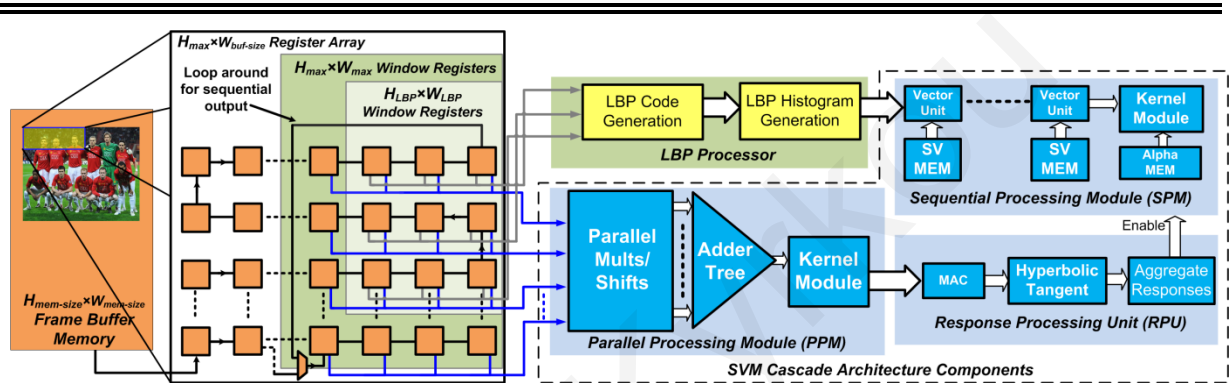


Figure 4-12. Support vector machine cascade system architecture

The architecture is comprised of the sequential processing module (SPM), the parallel processing module (PPM), the register array, frame buffer memory, the LBP processor and the response processing unit (RPU).

- *Parallel Processing Module (PPM)*

The parallel processing module (PPM) handles the processing of the low complexity SVM stages which have been adapted using the proposed hardware reduction method. Specifically, the proposed architecture can process linear and 2^{nd} degree polynomial kernels, but the plug and play approach of the architecture means that other kernel modules implementing different kernel functions can be used instead. The characteristic of the early cascade stages is that they require processing only a few SVs per input vector. Furthermore, they will have to process the majority of input vectors. As such, parallelism focuses on processing vector elements in parallel to reduce the processing time per vector.

The architecture of the PPM is comprised of three main regions (Figure 4-13): SVM shift operations, adder tree pipeline and kernel computation. The first region is comprised of parallel SV data memories, arithmetic shifters and parallel sign conversion units. The second

region is comprised of a tree of adders that sum the results of the previous stage in order to calculate the dot-product scalar value. The final region is dedicated to kernel processing and is also mostly implemented using arithmetic shift units.

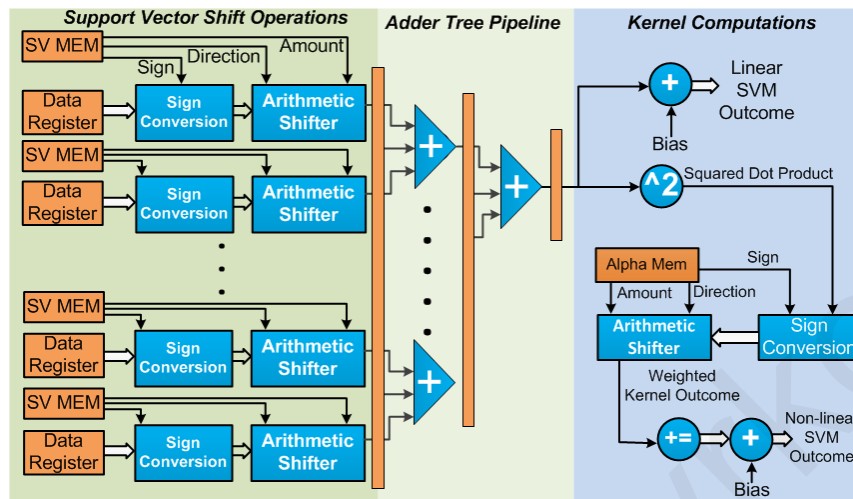


Figure 4-13. Parallel processing module (PPM) architecture

The PPM handles the processing of the nearest power of two adapted SVM stages. The shift units and adder tree are used by all kernels while only non-linear kernels use the rest of the kernel module.

The operation of the parallel processing module begins with the processing of the input vector elements by the sign conversion units which are used to preserve the sign of the initial multiplication operation. The signed numbers are then processed by arithmetic shifter units which perform the shift according to the data that they receive from the ROMs. The shift data are fetched in parallel from small ROMs, and include the sign of the support vector used to convert the input vector element to a positive or negative two's complement format, the shift amount, and finally the direction of the arithmetic shift operation. The partial results are added together using a pipelined tree of adders so that the dot-product outcome can be obtained. The length of the adder tree impacts the latency of the PPM and depends on the number of operands of individual adders used and the vector dimensionality. The latency of the adder tree is thus given by:

$$adder_tree_stages = \lceil \frac{\log(vector_dimensionality)}{\log(adder_input_size)} \rceil \quad \text{Equation 4-6}$$

Once the dot-product scalar value becomes available the kernel computation follows. In the case of linear kernels (Equation 4-1), adding a bias value to the dot-product outcome will suffice in order to obtain the classification result. However, for 2nd degree polynomial kernels, as well as other kernels, the kernel computation module handles the latter steps of the classification phase. Only one multiplier is used in the parallel processing module and is used to perform the square operation. The processing of the alpha coefficients is done with a sign conversion unit and an arithmetic shift unit similarly to the processing of the SVs. An accumulator is used to accumulate the result of each SV processing, and once all SVs are processed, an adder is used to process the bias with the accumulated result. The PPM stages are pipelined, so one SV enters the pipeline every cycle. Hence, the total number of cycles needed to process the input vector at stage n are given by Equation 4-7, where $N_{SV}(i)$ is the number of support vectors that need to be processed by stage i .

$$\left(\sum_{i=1}^n N_{SV}(i) + \text{adder_tree_stages} + 1 \right) \quad \text{Equation 4-7}$$

The PPM architecture describes a fully unrolled implementation and allows for all vector elements to be processed in parallel, thus providing higher detection speeds. In cases where the resources are not available or the vector elements cannot be accessed in parallel, the PPM architecture can be implemented using fewer resources, to meet the given constraints, by processing fewer vector elements in parallel.

- *Sequential Processing Module (SPM)*

The sequential processing module (SPM) is responsible for performing the processing necessary for the final SVM stage. This final stage will most likely process only a small percentage of the input data, however, it will have the largest number of SVs. As such, instead of processing the input vector in parallel the focus is on processing more support vectors in parallel. This is achieved with the architecture shown in Figure 4-14, which is comprised of a series of pipelined processing and memory elements. The majority of the units in the module are vector processing units (VUs) and each unit handles the dot-product for one support vector with the input vector. They are comprised of a multiply-accumulate unit, and also a ROM which contains the data for one or more support vectors, along with register and multiplexer

logic for data transfer between vector units. The final unit in the pipeline is the kernel processing unit which is equipped with multipliers and accumulators to carry out the scalar operations of the SVM processing flow.

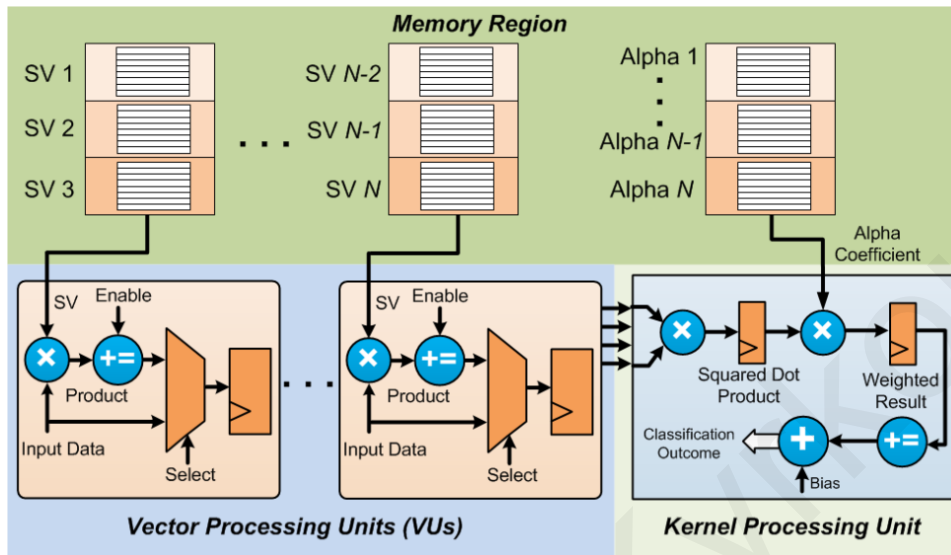


Figure 4-14. Sequential processing module (SPM) architecture

The architecture consists of two processing units: The dot-product processing units handle the dot-product computation, and the kernel processing unit, which is shared amongst the dot-product units, handles the kernel-related operations.

The input vector is processed with a group of support vectors at a time, and each vector processing unit handles the processing of one support vector. Once a group of support vectors is processed the next group follows. In total depending on the number of groups a total of $\lceil N_{SV}/num_of_VUs \rceil$ processing repetitions are necessary. Hence, the size of the pipeline can be adjusted to fit the available resources by adjusting the number of support vector groups. Each vector processing unit in the pipeline processes one support vector with the input vector at a time. The data in the SPM flows in a systolic manner as the input vector values are propagated from one unit to the next, through the dedicated transfer mechanisms, while the ROMs feed each VU with SV data in parallel. When the processing of the input vector with the group of SVs is done, the multiplexers and registers in each vector unit are used to switch from propagating input vector values to scalar results. The scalar values are transferred sequentially through the pipeline and are processed by the kernel processing unit (with a 2

cycle initial delay due to the pipeline stages). In this way the kernel processing unit is shared between the units, reducing hardware requirements and also making it easy for the designer to substitute it with the desired kernel without having to change much of the system functionality. Each scalar value that enters the kernel unit is processed by the kernel operation and the alpha coefficient. In the case of kernel (3) the operation involves a multiplier to find the square of the value and multiply-accumulate units to process the alpha coefficients. Once all scalar values are processed, the final classification result is obtained by adding the bias to the accumulated result. Overall the number of cycles needed to process an input vector is given by:

$$\lceil N_{SV}/num_of_VUs \rceil \times (vector_dim + num_of_VUs + 2) \quad \text{Equation 4-8}$$

- *Response Processing Unit (RPU)*

As previously described the objective of the cascade response evaluation process is to remove samples prior to the final SVM classification in order to improve processing speed. However, this needs to be done in a hardware efficient manner so that performance is not negatively impacted and not many resources are required. Hence, computationally and memory intensive algorithms are not the desired choice. For this reason a compact neural network (NN) is selected to perform the response evaluation which is computationally efficient and more importantly manages to sufficiently differentiate between cascade responses.

The neural network model is shown in Figure 4-15-(a). It is a two layer structure with a single neuron in each layer. The first neuron receives the responses from the cascade stages and multiplies them with their respective weights and accumulates the products. Then it adds the bias value and sends it through a hyperbolic tangent activation function and to the output neuron. The output neuron performs the same process and generates the classification outcome.

The neural network hardware architecture Figure 4-15-(b) processes cascade responses produced by the PPM. Since each response is generated at different time intervals, it can be processed once it is available by the PPM, so each response is processed sequentially. Multiplexers are utilized to select the output of the desired classifier and its corresponding

weight value, which is represented in a fixed point format. The two values are multiplied and accumulated. Once all the cascade responses are accumulated the *bias* is processed. A Look-Up Table (LUT) memory is used to implement the hyperbolic tangent function while exploiting the facts that this function is symmetric with respect to negative and positive inputs, and that its results range from $[-1..1]$. Consequently, only the results for positive numbers are stored with the input being processed to obtain its absolute value. This leads to a more compact and efficient implementation. The sign of weighted accumulated sum is used to adjust the result of the hyperbolic function memory after the appropriate value is loaded. Then it is processed with the output layer weight which is implemented using an arithmetic shift unit. Finally, the bias is added and the final outcome is computed. It is not necessary to use a hyperbolic function for the output layer neuron since it does not change the sign of the result, which determines the class, thus reducing the memory requirements. The RPU takes $(NN_pipeline_stages + 2)$ cycles to process the response vector that is generated from the PPM.

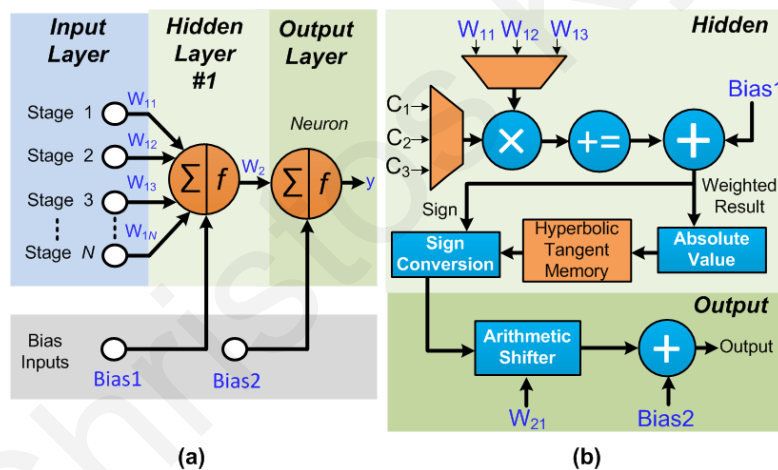


Figure 4-15. Response processing unit (RPU)

(a) Neural network model (b) NN-based RPU Hardware Architecture

- *Local Binary Pattern (LBP) Processing Units*

Local Binary Patterns (LBPs) [62] describe the relationship between a pixel and its neighborhood, and have been used in a wide range of computer vision applications [70]. Their major advantage is their low computational complexity [116]. Generating the LBP [70]

consists of the following steps: 1) Compare the values in a 3×3 neighborhood against a threshold (the center pixel or the window mean value) placing 1 where the value is greater or equal, and 0 otherwise. 2) Multiply the resulting binary map with a powers of two mask. 3) Sum the values to obtain the LBP Code. 4) Divide the LBP-based image into k blocks of $i \times j$ pixels (e.g. 4×4 , 8×8) and construct a local histogram of l bins. 5) Concatenate the local histograms to form a single global histogram descriptor.

The LBP descriptors can be used as features by the latter stages which have to process fewer data samples but require better discrimination capabilities. Since only a fraction of samples will be processed using LBP, it must have low area overhead. This is different to works such as [116] where the goal is to parallelize the LBP operations and thus dedicate more resources for LBP processing. Accordingly then, the developed LBP processor architecture features parallel processing of the values of only a single 3×3 window from the input image located in the register array (Figure 4-16-(a)). Those values are then loaded from the register array and each window value is compared against the center window value in parallel and the results are concatenated to generate the LBP code. This was preferred instead of using the mean-based threshold which in [116] which involves multiplication/division operations and adder trees to compute the mean, and would consume more resources. The number of transitions in the LBP code is found next. This is necessary to identify a pattern as uniform LBP code (which has 2 or less transitions e.g. 11110000) or non-uniform LBP code (which have more than 2 transitions e.g. 10100101). This offers a more meaningful interpretation of the LBP codes which can achieve higher discrimination. One LBP code is generated from a 3×3 window per cycle.

The histogram generation circuit receives the transition count for each LBP code and assembles the local histograms which are stored in the same central memory (of size $k \times l$), a process further illustrated in Figure 4-16-(b). The histogram counts the uniform LBP codes against the non-uniform. It has been shown [62] that uniform patterns account for the majority of image information. Thus, the histogram is split between uniform and non-uniform patterns where all non-uniform patterns are stored in one bin and each uniform in a separate bin. The first step in the histogram computation is to find the starting address for the local histogram which the LBP code belongs. This is achieved by counting the row and column of each LBP

code. By keeping track of the MSBs of the row and column coordinates it is possible to identify the block which it belongs to. Then by setting the appropriate address offset the corresponding histogram region is selected. The actual bin in the histogram is found using a LUT which maps the LBP code to one of 59 possible histogram bins. A dual ported memory is utilized to store the histogram. In this way an immediate reset can be performed right after the value is sent to the SVM from the second port which receives the same address but delayed by a single cycle.

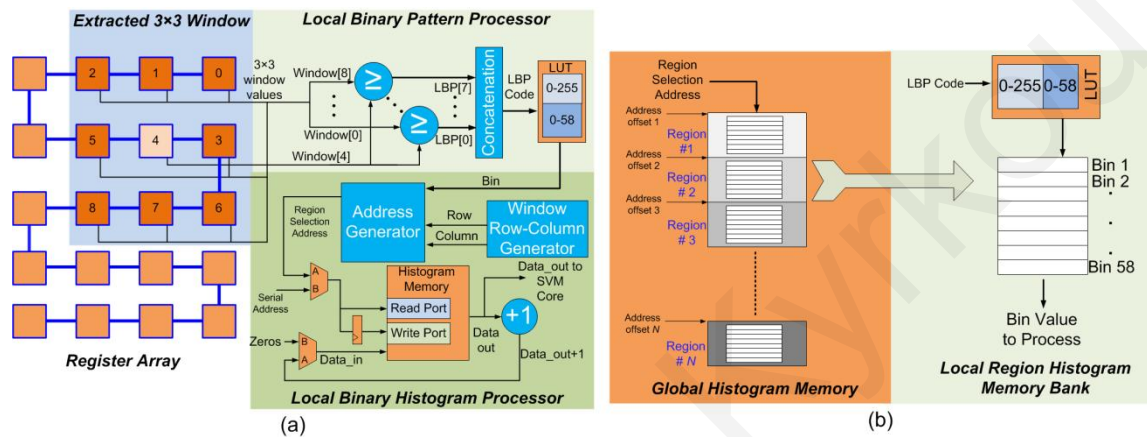


Figure 4-16. Local binary pattern processing hardware

(a) Local binary pattern (LBP) processor architecture (b) Histogram generation process

▪ *Cascade Processing Flow and I/O*

The different throughput requirements of the cascade SVM processing modules require an I/O mechanism that can adjust to the different needs of each module, that is parallel as well as sequential data transfer. It should also take advantage of the application-specific characteristics to facilitate data reuse and reduce memory accesses. Furthermore, different classifiers may utilize different data points or need to preprocess the data. The cascade I/O structure should be able to handle this. To illustrate the above features first, the design of such a structure for object detection applications is considered. An optimized I/O mechanism for object detection can be developed based on an array of shift registers that incorporates the above features and also acts as local storage for the image segment that is currently being processed (Figure 4-17). The register array has a size of size $H_{max} \times W_{buf_size}$, where H_{max} is the height of the maximum window and W_{buf_size} corresponds to the width of the array, i.e.

how many additional image columns are stored. The input image pixels enter the register array and are propagated row-wise into the structure. The image region that is at the left-most part of the register array corresponds to a $H_{max} \times W_{max}$ window and each unit receives data from specific registers that window. For example, the LBP processor receives 9 pixels from the left-most 3×3 window ($H_{LBP} = 3, W_{LBP} = 3$) to produce a $H_{max} - 1 \times W_{max} - 1$ image made up of LBP features. The PPM can receive register data corresponding to either a $H_{max} \times W_{max}$ window or any other downscaled version (e.g. a $(H_{max}/2) \times (W_{max}/2)$ window) if deemed necessary, by selecting the appropriate registers, thus achieving dynamic downscaling of the larger $H_{max} \times W_{max}$ window, referred to as $H_{min} \times W_{min}$ window. In this data flow the image region is processed in a window-by-window fashion. Once, a window has been processed a part of it is shifted out of the array, while new pixels are shifted in, thus a new window is formed at the leftmost region of the scanline buffer and is ready to be processed next. The data flow of the left-most registers changes depending on whether the data are used for parallel or sequential processing. In the case of the parallel processing module, window data are outputted and processed in parallel. In the case of sequential processing, which happens when the LBP features are generated, the registers form a chain so that data are outputted sequentially from the leftmost top row register. Furthermore, during sequential output operation, the window data are looped back to the scanline buffers, using a multiplexer on the start of the chain (Figure 4-17). This is required so that the window is formed again and placed correctly with respect to the rest of the image in the register array to maintain consistency.

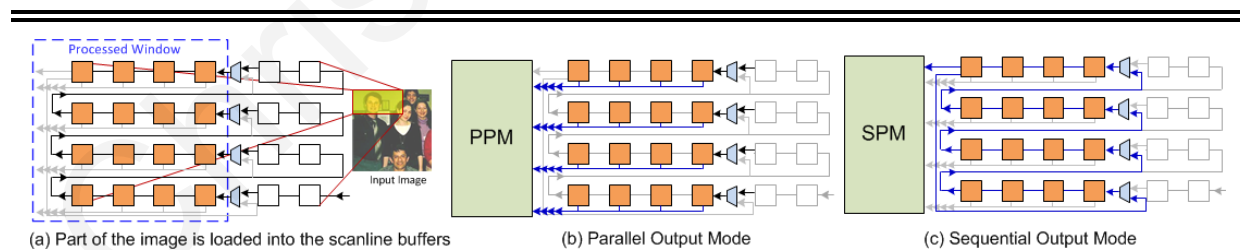


Figure 4-17. Optimized I/O mechanism

- (a) The array stores part of the image to be processed and outputs a window. The array operates in two modes: (b) The parallel output mode where all window pixels are outputted in parallel and (c) The sequential output mode where the window pixels are outputted serially and looped back to the array using a multiplexer.

4.3.3 Experimental Platform and Results

The proposed hardware architecture and methods were evaluated using the embedded application of face detection considering 800×600 (SVGA) resolution images. It was evaluated in terms of frame-rate, detection accuracy, power consumption, as well as requirements in terms of computing resources. The cascade structure, illustrated in Figure 4-19, was trained using MATLAB R2010b [46] and was used for evaluation of the architecture and proposed framework. Additionally, the proposed hardware architecture, which will be referred to as the *adapted cascade*, is compared against a *baseline system* which implements the same cascade SVM structure, but without applying the hardware reduction method, and thus the parallel processing module is implemented using multipliers. Both implementations were evaluated and compared using a Xilinx Spartan-6 Industrial Video Processing board equipped with a Spartan-6 XC6SLX150T FPGA [88] using Xilinx ISE 12.4 and EDK 12.4. A Microblaze-based [144] video-pipeline system was used for I/O and verification purposes, while for both systems an on-chip buffer is used to store the input image and a 60×20 register array for data loading and processing (Figure 4-18), which was experimentally found to provide an adequate between balancing I/O delays and hardware resources.

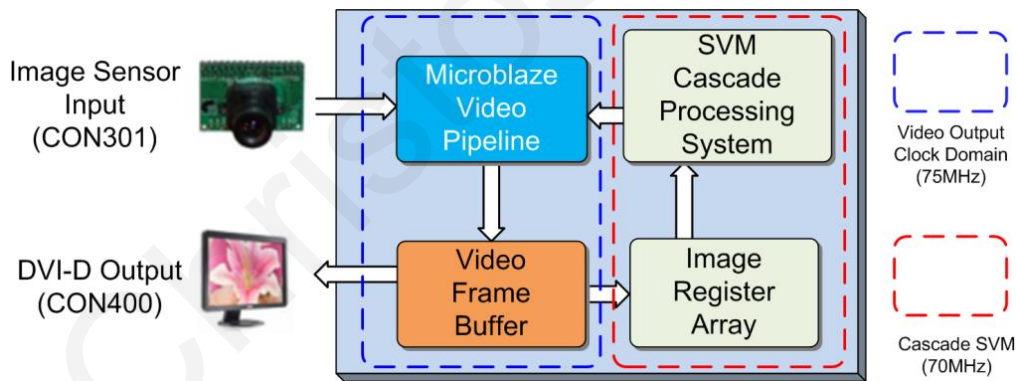


Figure 4-18. Block diagram of the FPGA system.

A. SVM Cascade Structure and Training

To evaluate the proposed hardware architecture and approaches an SVM cascade structure was designed, with kernels, and parameters similar to what has been used in the literature [54], [55], [56] and outlined in Table 4-5. The early stages tasked with the fast rejection of samples

correspond to linear SVMs or non-linear kernels with low computational demands [54]. Hence the cascade (shown in Figure 4-19 along with the parameters for each stage) used for evaluation purposes was comprised of two linear kernel SVMs and two 2nd degree polynomial kernel SVMs. The response processing acts as an intermediate stage between the adapted and final stages. The cascade structure processes 20×20 windows extracted every 5 pixels, which is similar to other works of cascade SVMs in the literature. This window size results in a 400-dimensional vector which is passed to the first three SVM cascade stages for rapid processing. Once a window has passed all three stages successfully its responses are given to the RPU which if it predicts a positive outcome the 20×20 window is processed to produce an 18×18 LBP feature image. Different experiments were carried out with all possible non-overlapping block sizes to find the one which provided the best accuracy results. This resulted in an LBP histogram descriptor with the parameters in Table 4-6. The resulting 1062-dimensional LBP histogram vector is passed as input to the final SVM cascade stage.

Face and non-face samples from [139] were resized to 20×20 pixels, resulting in 400-dimensional vectors, and used to setup an initial training set, which was later enhanced with additional samples (total of ~6000 positive and ~50000 negative samples). The three first cascade stages were trained using MATLAB, in incremental fashion [54], [56], [58]. The first stage was trained on the initial training set from [139] and adapted using the hardware reduction method. Then, the initial training set was enhanced with negative samples that were misclassified by the first stage, and the new training set was used to train the second classifier, and the same was done for the third stage. The final SVM stage was excluded from the process and was trained using the complete training set which was first processed using the LBP feature extraction. The first polynomial SVM (Stage 3, Figure 4-19) was reduced to 20 RSVs which was the smaller number of reduced vectors in order to maintain the original accuracy. In contrast 100 RSVs were needed to maintain the accuracy for the final stage (Stage 5, Figure 4-19) [56], [58]. The three first stages retained similar accuracy level after being rounded-off to the nearest power of two, as shown in Figure 4-21. However, for the final SVM there was a significant discrepancy between the classification accuracies of the adapted and original model. Hence the final SVM was not approximated and was implemented using the SPM architecture.

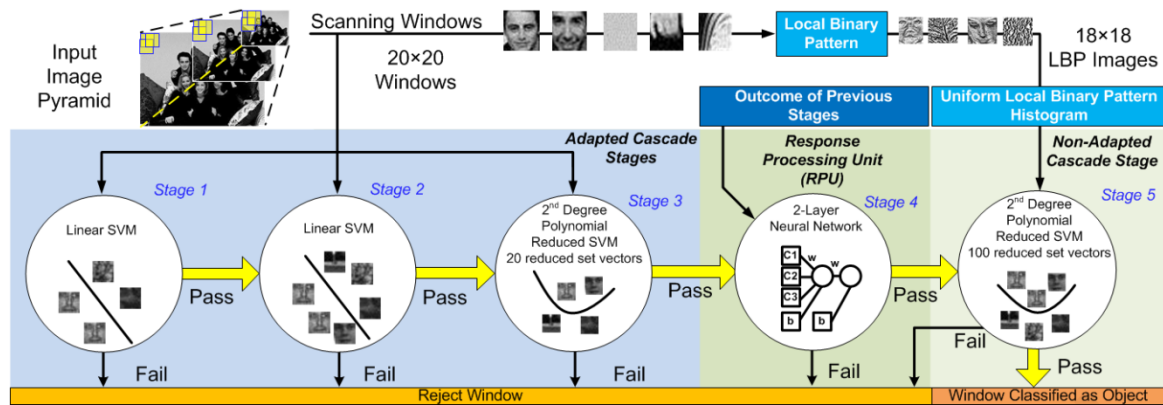


Figure 4-19. Cascade support vector machine structure

The first three stages have been rounded-off to the nearest power of two values and process 20×20 windows. The final SVM stage remained unchanged and processes the LBP feature histogram. The intermediate stage is a NN-based RPU and processes the responses of the adapted cascade stages.

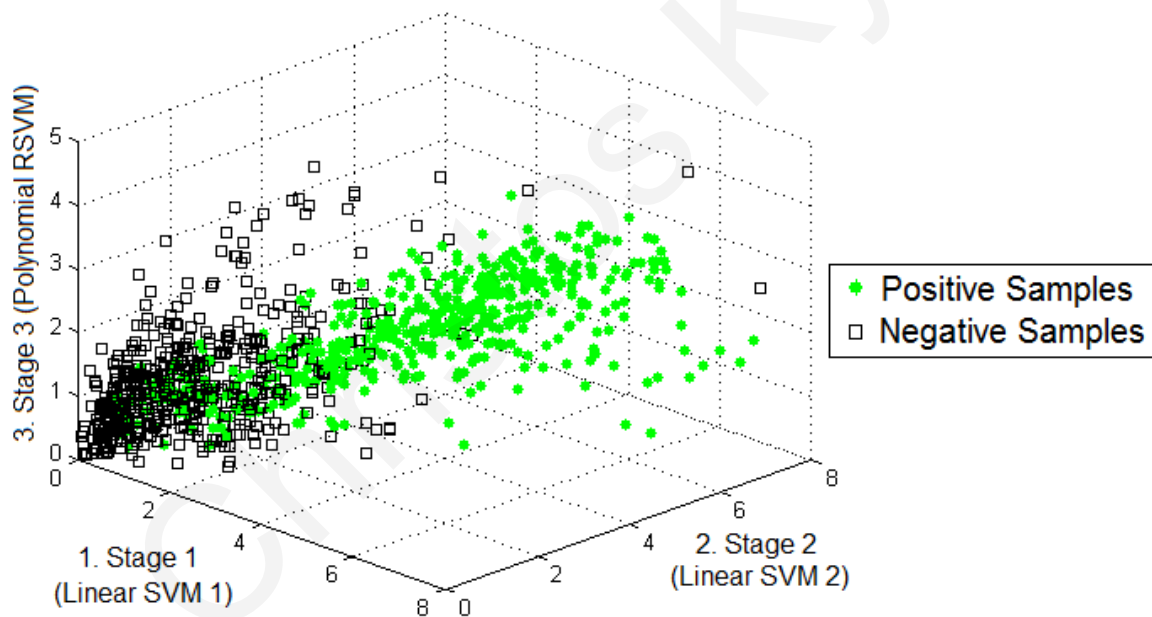


Figure 4-20. Support vector machine cascade early stage responses

Responses of the first three SVM cascade stages for 500 negative (square) non-face and 500 positive (filled circle) face samples. Each axis corresponds to the response for each classifier.

TABLE 4-6: CASCADE DETECTION SYSTEM PARAMETERS

Search Window Size	Downsampling Rate	Window Step	Image Resolution
20×20	1.2 (18 scales)	5 pixels	800×600 (SVGA)
LBP Block Size	Number of LBP Blocks	LBP Histogram Bins	Number of Windows
$i = 3, j = 6$	$k = 18$	$l = 59$	~56984

After the SVM cascade training, the training of the NN-based RPU followed using the process described in Figure 4-11. Additional 20×20 images of face and non-face samples, not used in the training phase, were cropped, resized and extracted from various images and were passed through the first three adapted SVM cascade stages to collect their responses. This resulted in a three dimensional response vector per sample. The response vectors corresponding to positive samples were selected to form a new training set. The cascade responses for a subset of this training set are shown in Figure 4-20, where it is evident that the responses of the early stages exhibit different patterns for positive and negative class samples. The NN-based RPU training resulted in a correct classification rate of 99% for positive and 60% for negative responses.

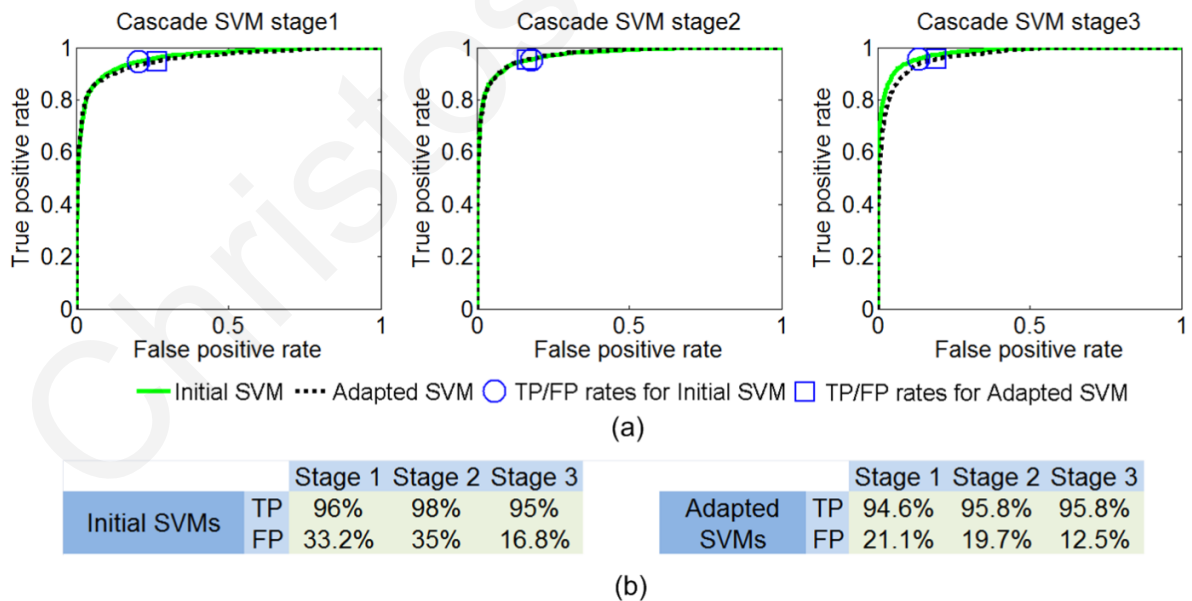


Figure 4-21. Adaptation using the ROC curve and new detection rates

(a) Adjustment of accuracy using the ROC curves (b) Accuracies before and after adaptation

B. FPGA Implementation and Resource Utilization

The two cascade implementations (baseline and adapted) have the same basic architecture (Figure 4-12) and data flow. The PPM architecture was based on a fully unrolled implementation, while the SPM was implemented with 50 DSP units ($num_of_VUs = 50$), meaning that the input data to the SPM is processed two times with different SV groups. Increasing this to 100 VUs, and process the input vector only once, can improve performance, however, the DSP utilization increases and so does power consumption. The NN-based RPU was mapped on the FPGA LUTs. The only difference between the two implementations is that in the adapted cascade case the PPM was optimized using the hardware reduction method. Consequently, the multiplication units were replaced with shift units and the data stored in the training data ROMs corresponded to shift values instead of support vector values. All the shift units were identical, even though not all support vector data require the full shift capabilities. As such, it is possible to exploit the flexibility of FPGAs in future iterations to design different shift units optimized for the specific requirements of support vector groups stored in the same ROM, resulting in fewer logic resources being utilized. Each ROM holds the support vector data for the first three cascade SVM stages for the specific vector elements. In the adapted cascade implementation 6 bits are needed to store the shift data: 4 bits for the shift amount, corresponding to a maximum shift amount of 15 bits, one bit for the sign of the support vector, and one for the arithmetic shift direction. For the baseline implementation 8 bits are needed to represent the decimal number SVs to maintain the same accuracy. In addition adder trees, used by the PPM and LBP processor, utilize ternary adders instead of two-input adders, to reduce the latency by a few cycles.

Both implementations utilize the same DSP and BRAM units since these were mapped to modules which are the same in the two implementations. These are the SPM, the RPU, and the LBP processor. Also, both implementations on the Xilinx Spartan-6 XC6SLX150T FPGA have the same critical path, the SPM kernel unit mapped on the DSPs, and as such have the same operating frequency of 70 MHz. The implementation of the adapted PPM requires 40% fewer FPGA logic resources compared to the baseline PPM. This is reflected with a 25% reduction in the utilized resources when considering full system implementations, as shown in Table 4-7.

TABLE 4-7: FPGA RESOURCE REQUIREMENTS PER UNIT AND SYSTEM

FPGA Resources	Registers (184304)	LUTs (92152)	BRAMs (268)	DSPs (180)
SPM	1736 (1%)	2241 (2%)	51(19%)	50 (27%)
Adapted PPM	2679 (1%)	19006 (20%)	1 (<1%)	---
Baseline PPM	3724 (2%)	30791 (33%)		
NN-based RPU	82 (<1%)	379 (<1%)	2 (<1%)	6 (3%)
LBP Processor	32 (<1%)	94 (<1%)	2 (<1%)	---
Memory & I/O Units	1831 (1%)	1200 (1%)	180 (67%)	---
Microblaze Video Pipeline	10780 (5%)	9891 (10%)	20 (7%)	3 (2%)
Baseline Cascade System	21214 (11%)	47396 (51%)	256 (96%)	59 (32%)
Adapted Cascade System	20153 (11%)	35532 (38%)		

C. Detection Accuracy and Frame-Rate

Accuracy and frame-rate are two important metrics in object detection and thus this section outlines these results. It also highlights the impact of the LBP processor and RPU on accuracy and frame-rate.

The accuracy of the adapted cascade SVM was evaluated on the JAFFE database of faces [146]. In addition these results were also verified on 200 images cropped and resized to 800×600 (SVGA) resolution from the CMU-MIT database [142], the Bao face database [147], and the world-wide-web. Full frame detection results are shown in Figure 4-22. The same set was used to evaluate the frame-rate of the cascade SVM implementations. Each 800×600 image generates a total of 56984 20×20 search windows for 18 scales and a window step of 5 pixels. Each frame requires a different time to be processed, by the cascade implementations, depending on how many windows reach each stage, and by how many cycles it takes a stage to process an input. All generated windows are processed by the first SVM stage, however, only $\sim 1\%$ of them reach the final SVM stage, as shown in Table 4-8. In addition to the actual processing time, the I/O delays per frame also negatively impact classification speed. In order to achieve higher detection rates, I/O and memory operations such as filling the register-array buffers, overlap with window processing.



Figure 4-22. Detection results on 800×600 images

(a) Detection Results using the RPU (b) Activity in the image: The darkness of the pixels indicates that the region has gone through more stages. (Some dark regions are formed by overlapping grey areas) (c) Detection Results without the RPU. Notice the increased false positives.

Performance metrics for different cascade configurations are shown in Figure 4-23. Specifically, Figure 4-23 shows true positive (TP) and false positive (FP) detection accuracies and average frames-per-second with and without using the LBP processor and RPU. The cascade SVM boosted by the NN-based RPU was able to achieve an accuracy of 80% which was only 1% less than the same system without using the RPU. The minimal drop in accuracy, when using the RPU, is offset by a $2 \times$ increase in performance. It allows the cascade system to operate at ~ 40 FPS instead of ~ 20 FPS, making the system capable of real-time performance. This happens because even though most windows are discarded by the first two cascade stages, the NN-based RPU manages to reduce the number of windows (~ 230 instead of ~ 715 , Table 4-8) that reach the slower SPM. Furthermore, the introduction of the LBP feature extraction process helped to improve both the true positive (TP) rate as well as the false positive (FP) rate, the latter by an order of magnitude, even though configurations that do not incorporate the LBP processor provide higher performance since the additional processing is not required and the feature vector dimensionality is reduced. Overall, the combination of LBP features and the RPU results in an adequate trade-off between frame-rate and detection accuracy with only a $\sim 2\%$ overhead in hardware.

TABLE 4-8: STATISTICS FOR EACH CASCADE STAGE

Cascade Stages	Stage 1 (PPM)	Stage 2 (PPM)	Stage 3 (PPM)	Stage 4 (RPU)	Stage 5 (LBP & SPM)
Number of Windows Processed	56984 (100%)	3025 (5%)	2334 (4%)	713 (1,2%)	228 (0,4%)
Rejection Rate	94,6%	22,8%	69,4%	76,4%	---
Cumulative Processing Cycles	9	10	30	35	2697
Vectors per stage $N_{SV}(i)$	1	1	20	---	100

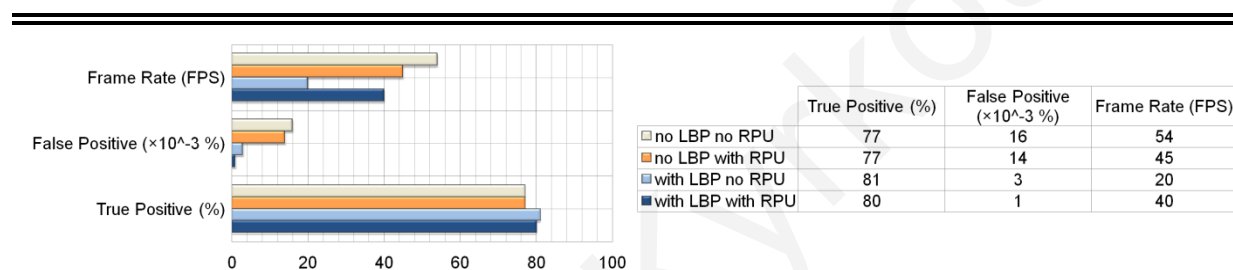


Figure 4-23. Comparative results of different cascade configurations

D. Power Consumption

Power analysis tools from Xilinx were used to measure power consumption demands of the adapted and baseline cascade SVM FPGA implementations. The characteristic of the cascade architectures is that the PPM and SPM are not used at the same time since they implement different cascade stages. Hence, the dynamic power consumption ranges depending on which module is active. The total power budget, including the Microblaze video pipeline, for the adapted cascade SVM system ranges from 4,1 W to 8 W while for the baseline cascade system it ranges from 4,1 to 9,9 W. The peak power consumption happens when the PPM module is used. The lowest consumption happens when the NN-based RPU is used when the SPM and LBP cores are used power consumption reaches 4,9 W. Overall, the utilization of less LUT resources by the PPM results in reducing the peak power needed for the system to operate by ~20%.

E. Summary

There are some useful conclusions extracted from the Spartan-6 FPGA evaluation. Overall, the proposed approaches improve various aspects of SVM classification problems such as the frame-rate through the parallel hybrid architecture and the response evaluation method (40 FPS for 800×600), the false detection rate through a compact LBP processor (by an order of magnitude), as well as power consumption and resource utilization through the hardware reduction method (by 20% and 25% respectively). In addition the compact implementations of the RPU and LBP processor added only a 2% overhead in logic resources, while there is only a 1% penalty in the detection accuracy.

TABLE 4-9: COMPARISON WITH RELATED WORK

Related Works		Rojas [119] ^a	Kryjak [116] ^b	Bauer [107] ^c	Presented Work
Application		Barcode Detection	Head-Shoulder Detection	Pedestrian Detection	Face Detection
Classification Methods		Polynomial SVM with 88 SVs	Linear SVM & LBP	SVM (GPU) & HOG (FPGA)	Cascade SVM & LBP
Platform		Xilinx Virtex II Pro XCV3000	Xilinx Virtex 6 XC6VLX240T	Xilinx Spartan 3 & NVIDIA GPU ^c	Xilinx Spartan 6 XC6SLX150T
FPGA Resources	LUT	22938/28672	12068/150720	28616/62208	35532/92152
	REG	N/P	15893/301440	N/P	20153/184304
	BRAM	160 KB	124/416	100	256/268
	DSP	N/P	66/768	18/96	59/180
Image Size		512×512	640×480	800×600	800×600
Window Size		16×16	32×24	48×96	20×20
Vector Size (<i>Vector_dim</i>)		256	1440	1980	400 & 1062
Number of SVs		88	1	N/P	122
Frequency		166 MHz	40 MHz ^b	63 MHz	70
Detection Accuracy		TP: 91,8% FP: 4,2%	TP: 83%	TP: 95,4% FP: 0,1%	TP ~80% FP: ~0,001%
FPS		N/P ^a	60	10	40

^a Performance is 352 cycles per sample just for the vector operations. No I/O delays are included.

^b Uses a frequency multiplier to multiply the clock three times for the SVM processing Core (120 MHz).

^c A hybrid system where the GPU implements the SVM and the feature extraction based on HOG is implemented on the FPGA.

N/P – Not Provided | N/A – Not Applicable | CD - Correct Detection | TP - True Positive | FP - False Positive

F. Comparison with Related Work

Related works for object detection applications are shown in Table 4-9, along with information regarding parameters and performance. These works use different algorithms, training and test sets, and benchmark applications, hence it is difficult to make a direct comparison between implementations. Nevertheless, a discussion is made to attempt to position this work against others and clarify its contributions.

SVMs have been used in various object detection applications and as a result FPGA implementations for SVM-based object detection have used different applications and parameters to benchmark the proposed architectures. However, since the SVM classification flow treats all data as vectors, the number of samples and SVs processed and vector dimensionality can provide an indication to the processing performance for each work. The number of samples depends on the search window size and granularity of the search. The different benchmark applications mean that the search window size and feature vector size are different. A head-shoulder detection system is presented in [116]. It utilizes an SVM and LBP descriptors to classify 19200 windows from 640×480 images. It trade-offs accuracy for performance by using a single linear SVM, with a clock frequency of 120 MHz, and processes only a few elements of the SV feature vector in parallel to keep the resource utilization low thus achieving 60 FPS. However, non-linear kernels often provide better and more robust results compared to linear kernels and thus might be the preferred choice for applications that require high accuracy, in which case more processing resources will be required to maintain real-time performance. In order to compensate for the accuracy of linear SVMs they use foreground detection to verify detection results. The implementation in [119] scans a 512×512 image in non-overlapping blocks to perform bar-code detection. It performs the dot-product operations in 352 cycles for one window however, the scalar operations are not included. Furthermore, it processes only around 1024 16×16 window samples, corresponding to 256-dimensional vectors, per image, and it does not downscale the input image which simplifies the I/O and memory accesses. The hybrid FPGA-GPU pedestrian detection [107] for 800×600 images achieves over 10 FPS for the classification of 1000 windows. The lower frame-rate can be attributed to the larger feature size, however, the number of processed windows is an order of magnitude less than in the present work. In

addition, the use of GPU may prohibit such implementations to be used in embedded applications due to power consumption constraints.

Overall, in order to achieve real-time performance existing works rely on processing a few window samples, smaller image resolutions, or process a few SVs. Through the proposed architecture and methods it is possible to process higher resolution images in real-time while also reducing the implementation requirements.

The SVM hardware implementations target different applications and thus accuracy is difficult to compare. Software based implementations [54],[55],[56] that utilize cascade SVMs for face detection achieve accuracies that range between 78 – 80% while utilizing similar training set sizes (Table 4-5). The proposed optimized SVM cascade system achieves a detection rate of 80% which is on par with other works.

G. Discussion and Impact

There are some useful conclusions extracted from the FPGA evaluation. Firstly, exploiting cascade information in the form of the responses of previous stages has shown great promise in improving the performance both in terms of frame-rate and false detection rate for SVM classification applications. In this regard it is anticipated that it can be used with other cascade structures in order to provide speedups as well. Furthermore, there is the potential for online training of the RPU using the final SVM stage as the supervisor to retrain and adjust the RPU based on run-time information. This opens up new possibilities to further improve its capabilities and to dynamically respond to different scenarios. Second, the proposed hardware reduction method provides an efficient way to reduce the required logic resources of the cascade which is easy to implement and can thus be widely adopted. Finally, the hybrid processing hardware architecture and flexible I/O structure demonstrated how it is possible to efficiently utilize the available hardware and provide the necessary performance, by optimizing the design for the specific data flow and processing demands of cascade SVMs. It is anticipated that the proposed architecture and methods can be used to both design low-cost fast SVM coprocessors to accelerate more demanding monolithic SVM classifiers, or optimize existing cascade SVM classifiers.

4.4 Conclusions

Support vector machines are a widely used state-of-the-art classification algorithm that has exhibited high classification rates for a wide range of applications including visual object detection. This chapter described the hardware acceleration of monolithic SVMs through a dedicated array-based hardware architecture. The presented architecture is able to provide real-time processing by parallelizing both support vector and input vector operations and is generic and can process different kernel functions. Through the array-based architecture different units communicate seamlessly and thus parallel access can be provided using a serial-in-parallel-out register structure, while the more demanding units can be shared amongst the simpler units to reduce hardware overheads.

The hardware acceleration of cascade SVMs was also considered, which can be used to design intelligent embedded visual object detection systems. The hybrid processing architecture takes advantage of the nature of the cascade classification problems, and along with a hardware reduction method and a novel response evaluation method, manages to achieve adequate trade-off between accuracy, performance, power, and resource utilization. Also useful methods were presented which can be used to design optimized hardware-efficient architectures with respect to constraints involved in embedded applications.

Overall, the research with regards to SVMs demonstrated how the development of array-based hardware accelerators adapted and optimized for the SVM processing flow can lead to real-time performance using FPGA platforms.

Christos Kyrkou

CHAPTER 5

REAL-TIME HARDWARE ACCELERATION OF OBJECT DETECTION USING DEPTH AND EDGE INFORMATION

The previous chapters have outlined FPGA-based object-detection hardware architectures which focus on the parallelization of the classification phase. The majority of these architectures employ a traditional sliding-search-window-approach to search for objects, and also downscale the image several times to find objects of different sizes. However, as the image resolution increases, the number of generated search windows increases as well depending on many factors including the number of scales that need to be searched (from highest to lowest resolution), the overlap between successive windows and the window size itself. This increase in search space can make it difficult to meet real-time constraints while being able to detect objects at different sizes. This is apparent from Figure 5-1, which shows how the number of windows increases with the image size and number of scales for typical object sizes from datasets [5],[139],[140] for different applications (face detection, pedestrian detection, car detection). Both these factors make it challenging to provide real-time processing. Furthermore, as the number of data to be classified increases the probability of a false detection also increases and also energy is wasted on processing windows that most probably do not contain an object of interest. It is possible to increase the window size as the image size increases in order to reduce the search space, however, in such a case, the classifier demands on hardware resources, memory, and processing speed will also increase. Thus it is preferable to keep the window size at a considerably small size and introduce techniques to compute the size of windows without having to exhaustively search the scale space.

Software implementations of object detection applications use background removal techniques such as motion detection [150] and color processing [151], [152], to reduce the search space. However, only a few hardware implementations feature such search-reduction methods that could potentially improve the efficiency of embedded object-detection systems [7], [72], [153]. Additionally, some of these techniques, such as color processing, are application/object specific and thus cannot be used in a variety of object-detection applications. Finally, search-reduction techniques that have been used in hardware do not

provide a way to identify the object size, and thus, exhaustive search must still take place even in a smaller image region.

Alternatively, with the emergence of 3D systems and algorithms and corresponding camera sensors [71] it is possible to utilize depth information to accelerate object detection. Depth information has been successfully used in software implementations for intelligent object recognition systems mostly to remove false detections [154], [155], [156] and to find object sizes [157]¹, however, many assumptions and simplifications (such as reduced search granularity) are made to allow for software implementations to achieve near real-time performance (11- 20 FPS). Furthermore, recently, edge information has been proposed as a medium to reduce the search space involved in object detection in software [158]. This chapter present research on how the preceding two methods can be merged together into a single algorithm and how that algorithm can be implemented in hardware utilizing a dedicated architecture in order to provide an efficient approach for acceleration of embedded object detection applications.

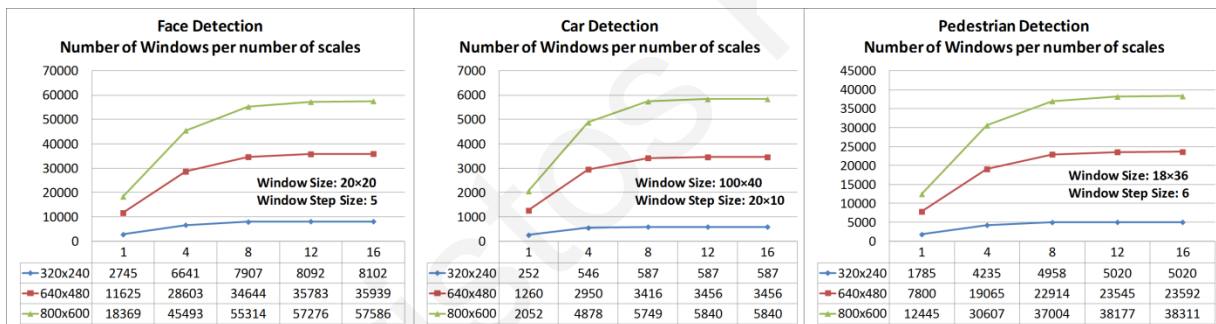


Figure 5-1. Number of windows as the image size and number of search scales increase

5.1 Depth- and Edge-Directed Search Space Reduction

Object detection is concerned with identifying the presence of an object of interest in an image. This is a tedious task which typically involves a sliding window scanning the input image and various downscaled versions of it in order to find objects of interest in various

¹ Pentium 4 (R) @ 3GHz, Bumblebee Stereo Vision Camera Point Grey Research

sizes. This exhaustive search makes it difficult to meet real-time constraints, especially as the image resolution increases, since more scales will need to be searched for the object of interest, and as a result, the number of search windows also increases. The remainder of this section outlines how depth and edge information can be utilized to reduce the search space and speed up the object-detection process.

5.1.1 Depth Extraction and Object-Size Estimation

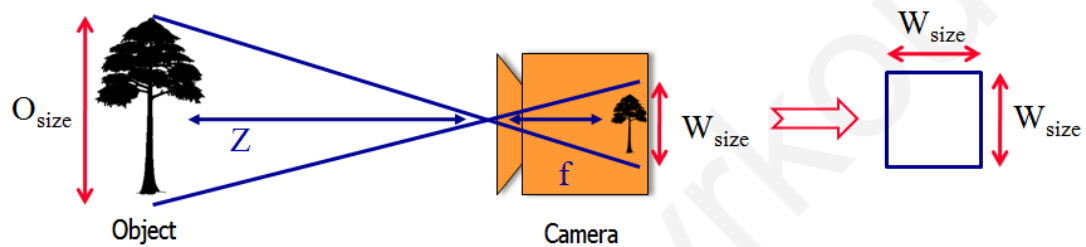


Figure 5-2. Window size estimation using depth information

Depth information (i.e., the distance of an object from the camera) can be extracted from the host environment of the embedded object-detection system. There are a number of methods that could be used to extract depth information from the host environment, such as the Microsoft® Kinect™ sensor or a stereo vision system that processes a stereo image (a pair of left and right images) [77]. In the context of stereo camera systems, information about depth (Z) evolves from the computation of the disparity map $d(x, y)$ using the formula $Z = f * (b/d(x, y))$, where b refers to the baseline distance between the stereo camera optical centers, and f refers to the focal length of the stereo camera system. As such, any stereo-based depth extraction method that can produce the disparity map could work in the context of the presented approach. By using depth information extracted from a vision system, it is possible to estimate the size of the object at a given location of the image, thus avoiding downscaling the input image several times and subsequently reducing the number of windows that need to be classified. The actual size of the object (O_{size}), as is represented in the real world, and its projection in one of the stereo image frames (W_{size}) can be estimated using Equation 5-1 [77]. The equation is derived from the arrangement shown in Figure 5-2, which shows that the ratio between the size of the object in the real world (O_{size}) and the distance from the camera

(Z) equals the ratio between the size in the image (W_{size}) and the camera focal length (f). Additionally, using the relationship between depth and the disparity map (d) that applies for stereo vision systems, the disparity value can be used instead (Equation 5-2), thus avoiding the need to compute the actual depth.

$$(W_{size}/f) = (O_{size}/Z) \Rightarrow W_{size} = f * (O_{size}/Z) \quad \text{Equation 5-1}$$

$$Z = f * (b/d) \Rightarrow W_{size} = (O_{size}/b) * d \quad \text{Equation 5-2}$$

5.1.2 Edge-Based Window Rejection Process

The performance of object-detection systems also suffers from the necessity that all windows need to go through the classification process. Thus valuable computational time as well as power are wasted on potentially unpromising regions. An efficient way to eliminate windows prior to the classification process, in a manner that can be parallelized in hardware and does not require many hardware resources, is utilizing edge information. Edges provide information about visual features in an image, and thus the number of edge pixels in an image can give an indication of the useful information in a particular image region. Hence, edges can be used to discard non promising regions (image regions that do not exhibit high pixel changes and thus there is a high probability that these regions will not contain any useful information), eliminating them from the detection process. The edges are extracted from the image by convolving the image with an appropriate edge mask (e.g. Sobel operator), and are then used to identify background and non-background regions. An example is shown in Figure 5-3.

An early approach to eliminating windows from the classification process, in addition to constructing the windows, was first proposed by Anila and Devajan [158]. The proposed algorithm relied on edge information to find boundaries of objects in order to enclose it with a window and extract that region. In the case where there were zero edge pixels within that region then that window was discarded from the classification phase and was classified as background. However, this approach was only tested on single face images with uniform background regions where edges are clearly formed only on object boundaries. However, the zero edge pixels threshold is susceptible to noise and is only able to remove regions with

almost uniform intensity. Hence, for cluttered backgrounds and noisy environments this approach would have diminishing returns since all windows would be considered for classification. A better approach would be to set an edge pixel threshold that is empirically derived from the object training set, by averaging the number of edge pixels found in object images. Finally, another potential problem with the original algorithm is that not all objects are enclosed within a boundary such as faces. Nevertheless, for certain applications and environments the edge-directed object detection processes can provide a computationally efficient way of reducing the search space involved in object detection.

Hence, in this research the window rejection process is altered to have a non-zero threshold, assigned according to the object of interest, and is combined with the depth-based window size estimation process into a novel approach in order to achieve both accurate window size estimation and rapid window rejection. The proposed algorithm is outlined in the following section.

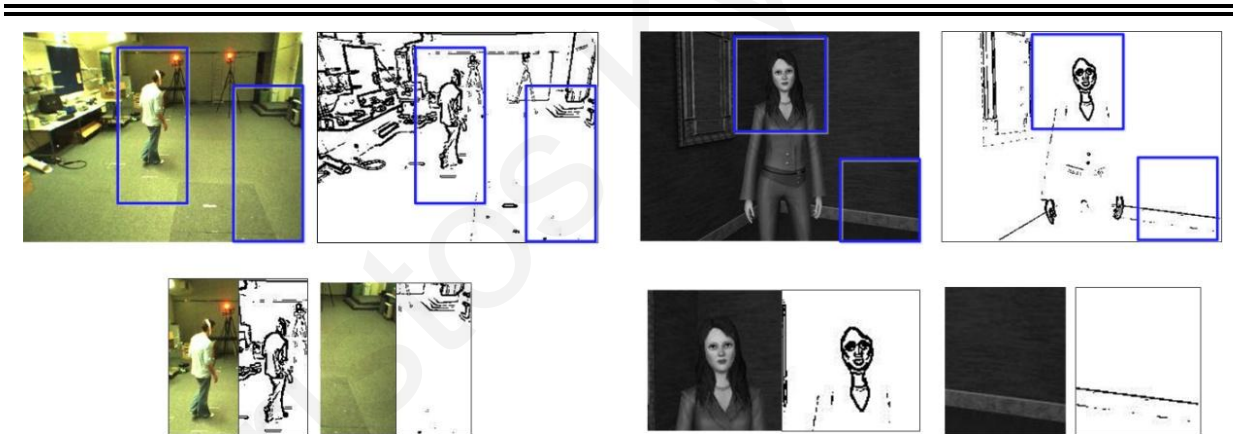


Figure 5-3. Object vs background edge information

(Top) Full Frame images with their respective edge image. (Bottom) Edge information for marked image regions

5.1.3 Depth- and Edge- Accelerated Object-Detection Process

An overview of the proposed accelerated object-detection approach is given in Figure 5-4. The approach relies on the computation of the disparity map from a stereo image to extract depth information. Each value in the disparity map corresponds to the center of a candidate

window. The candidate window size is determined by Equation 5-2, and a candidate window is formed around each disparity value. The disparity map is sampled every few pixels (disparity search overlap), and a candidate window is extracted for the sampled disparity values only rather than for every pixel in the disparity map. Furthermore, if the size of the candidate window exceeds the image boundaries, indicating the presence of a large object at the border of the image, that window can be discarded, as the object may fully fit in another candidate window. Finally, any disparity values that map to window sizes which are smaller than the size of the classifier can also be discarded on the premise that they wouldn't be considered for classification in the sliding-window approach either. After the size of the candidate window has been validated, that window is extracted from the edge image, resized to the search window size, as shown in Figure 5-4, and the number of edge pixels contained in the window is determined. If it exceeds a predetermined threshold, the candidate window is considered valid, and that window is extracted from the grayscale image, resized to the search window size, and then classified by the classification algorithm. Otherwise, it is discarded and the window extraction process is then repeated for the next disparity value.

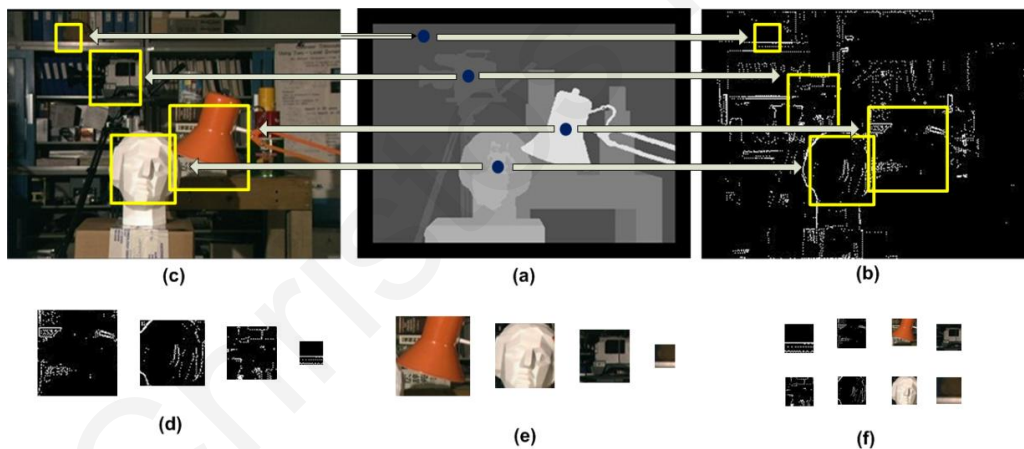


Figure 5-4. Object detection process using edge and depth information

Depth- and edge- accelerated process: (a) The disparity map is sampled every few pixels. Each disparity value corresponds to the center of a candidate window. (b) If the window size is valid, then the edges of that window are extracted from the edge image. (c) Corresponding window sizes in the original image. (d) Extracted edge windows. (e) Corresponding extracted windows from the original image. (f) Resized windows used for the calculation of the mean number of edge pixels and classification, respectively.

5.2 Hardware Visual Object Detection Systems

In recent years, a fair amount of work has been done on development of algorithms and techniques used in 3D vision systems. For example, [159] uses local feature histograms extracted from range images to recognize single object images. Both 2D and 3D features from color and depth images are used in [160] respectively, in order to recognize objects. Affine feature finders are combined with SIFT in [161] in order to provide robust 3D recognition under viewpoint changes and lighting and scale changes. More recently, efficient 3D feature-extraction algorithms have been proposed in the literature, such as the Fast Point Feature Histograms (FPFH) used for 3D image registration [162] and Normal Aligned Radial Features (NARF) that are used to find and recognize 3D objects in range images [163]. Finally, other approaches, such as [164], use generated 3D object representations to recognize different objects. Common classification methods that have been used in 3D object-recognition methods include nearest-neighbor methods [164], multilayer perceptrons [160], and support vector machines [165]. The main focus of these works was on providing higher recognition accuracy that could be used in a generic framework for 3D object recognition. However, hardware implementations of such 3D object-recognition systems have been rather limited to simple problems with low-resolution images [121]. Nevertheless, the advancements in the research of 3D vision systems provide an opportunity to utilize additional available information, such as depth, in order to accelerate 2D object-detection systems by reducing the search space.

A fair amount of work has been done in hardware implementations of 2D object-detection algorithms, mostly utilizing FPGAs and targeting face detection. These work use appearance-based methods that either employ pattern-recognition algorithms for the classification stage, such as neural networks, support vector machines, or a boosting learning approach (AdaBoost by Viola and Jones) and utilize the sliding-window approach to search for objects of various sizes. The following illustrates some of the most important works found in the literature, while Table 5-1 provides a summary of the techniques used in each work and the achieved performance.

TABLE 5-1: SUMMARY OF FPGA IMPLEMENTATIONS OF OBJECT DETECTION SYSTEMS

Work	Algorithms	Image Resolution	Classifier Window Size	Frames Per Second	Accuracy
He [7]	Skin Detection, Motion Detection Haar-Features and Neural Network	640×480 downscaled to 80×60	11×11, 19×19, 27×27	625	N/P
Han [166]	Modified Census Transform, Viola Jones	320×240	N/P	149	99.76%
Sadri [72]	Neural Network, Skin Detector	800×600	18×22	9	N/P
			or 36×44	90	
Cho [84]	Viola-Jones Detection Framework	320×240	20×20	23	N/P
Hiroto [83]	Viola-Jones Detection Framework	640×480	24×24	30	N/P
Farrugia [167] ^a	Convolutional Neural Network Face Finder	320×240	32×36	127 ^a	TP: 87%
		640×480		29	
				35 ^a	
McCready [168]	Neural Network	320×240	30×32	30	87 %
Presented Work	Depth Extraction, Edge Detection, Support Vector Machines	320×240	20×20	271	TP: 82% FP: < 1%
		640×480		42	TP: 82% FP: < 1%
		800×600		23	TP: 80% FP: < 1%

^a 127 and 35 FPS are achieved with a 25 PE ring on a Virtex 5 LX330. No utilization results are given.

N/P - Not Provided | TP - True Positives | FP - False Positives

One of the first attempts at implementing face detection in hardware was that of McCready et al. [168], and it was the first to demonstrate the benefits from a dedicated hardware solution. The authors designed a custom face-detection algorithm based on a neural network classifier and optimized it for the TM-2 (Transmogripher 2) configurable multiboard FPGA platform. It operated on a 320×240 image frame and achieved a frame rate of 30 FPS while having a detection accuracy of 87%. However, the proposed implementation utilized nine boards on the TM-2 system.

The detection framework by Viola and Jones [15] is a widely used approach for 2D object detection. The main benefit of this algorithm in terms of hardware implementation is that it

only requires additions and only a few multiplications, making it attractive for resource-constrained systems. Hence, a few hardware implementations of this algorithm can be found in the literature. Hiromoto et al. [83] proposed a hybrid model that consisted of parallel and sequential modules. The most frequent parts of the algorithm are implemented by the parallel modules, while the least frequent by the sequential. Through this approach, the authors manage to save hardware resources while achieving 30 FPS. Cho et al. [84] showed a parallel implementation of this algorithm on an FPGA, demonstrating near real-time performance of 23 FPS for 320×240 images.

Han et al. [166] proposed a face-detection system that utilizes the Modified Census Transform (MCT) and the AdaBoost learning algorithm with cascaded classifiers targeting mobile environments. They use only a single classification stage, and their system can detect up to 32 faces. Their system achieves high frame rates and high detection accuracies, however, the FPGA architecture and FPGA utilization information are not presented, making it difficult to assess their proposed system.

The hardware implementation of a convolutional face finder [169] based on a convolutional neural network was demonstrated in [167]. The proposed architecture consisted of a ring of processing elements (PEs) that implemented the CFF algorithm and demonstrated how pipelined architectures can be used to speed up the detection process. They exploit parallelism by dividing the image in vertical strips with overlapping regions, and each PE processes a block of that strip. However, a 25-ring PE is required to achieve a real-time performance on 640×480 images, and consequently, a large FPGA is used to fit the architecture. Furthermore, not enough details are given on the I/O requirements of their architecture and the memory and buffering requirements.

The aforementioned works have demonstrated how dedicated hardware solutions and classifier optimizations in hardware can provide high speed ups and real-time performance. However, higher frame-rates can only be achieved by integrating hardware search-space-reduction mechanisms. The following works demonstrate different methods and approaches that have been implemented in hardware in order to reduce the search space.

Ming and Yisong [153] use a hardware/software approach to design an FPGA-based face-detection system with skin-detection acceleration. A Nios-embedded soft processor handles

the face identification task, while a dedicated hardware accelerator handles the skin-detection process. This specific work focuses on accelerating the search-space reduction in hardware rather than the actual classification. The processor can then handle the classification process, since the search space is reduced. However, this also depends on the input image size.

Sadri et al. [72] implemented a face-detection neural-network algorithm on a Virtex II Pro FPGA, which included a skin color filter to reduce the search space within the image and an edge-detection mechanism to produce a binary image on which the neural network operates. The majority of the system was implemented on the FPGA custom logic fabric, while higher-level operations were left to the Power PC processor. Their system operated on 800×600 input images with a frame rate of 9 FPS if the entire image was processed. The resulting frame rate could be improved up to 90 FPS if only 25% of the image was searched, using skin detection, and only up frontal detection was considered. The approach of using binary-image data demonstrates the potential performance benefits, however, the impact on detection accuracy is unclear.

Finally, He et al. [7] demonstrated the hardware implementation of a massively-parallel face-detection system that achieved frame rates of over 600 frames per second. They utilize two search-reduction techniques, motion detection and skin detection. They also make a few simplifications on the face-detection procedure and adding two search-reduction techniques. The input image is of 640×480 pixels, however, all subsequent operations, including the detection process, happen on a downscaled 80×60 image, greatly reducing the search space, while only considering three face sizes (11×11 , 19×19 , and 27×27). The classification for the three window sizes is done in parallel and thus the proposed system needs a lot of resources to provide these high frame rates. This approach may not be suitable for other object-detection applications, as downscaling the input image may result in loss of quality.

These works have demonstrated the successful implementation of object-detection systems on FPGAs. The majority of these implementations have targeted face detection applications using the sliding-window approach or integrated face-detection-specific search-reduction techniques, such as skin detection. The majority of these works have targeted image sizes of 320×240 pixels for face detection, however, other applications may require processing higher-resolution images. Recognizing that there is a need for generic object-detection and with the

advancements in the field of 3D vision systems, an object-detection methodology and a hardware architecture are proposed that is based on depth as well as edge information that can provide a more generic platform for various detection applications and can facilitate real-time processing of larger image sizes.

5.3 Hardware Architecture for depth- and edge- based object detection acceleration

The architecture that implements the depth and edge directed object-detection process consists of four major hardware units, each implementing the major tasks of the algorithm, as outlined in Section 5.1.3. These units are the Disparity Computation Unit (DCU) which computes the depth information from a stereo image, the Edge Computation Core (ECC) that implements a Sobel edge detector, the Window Extraction Unit (WEU) that processes the depth and edge information and extracts windows, and the Classification Engine (CE) which implements a support vector machine classifier. It is also possible to use different approaches and algorithms for each of these units that meet application specific constraints for accuracy and power as well as performance. The specific approaches for depth and edge were selected since they provided high performance at a low cost of processing resources. The system also consists of a memory controller and a system controller that optimizes accesses to the external memory, control I/O operation, and synchronize the other major units. The system uses three on-chip buffers to store the image data, there are dedicated buffers for the edge image, the disparity image, and the left image of the stereo pair, which is used as the reference image for the disparity computation. Figure 5-5 shows an overview of the system architecture and the communication flow between units. Certain features of the architecture were optimized for FPGA implementation, however, the architecture can also be implemented using an ASIC design flow with only minor adjustments.

The whole process begins when the memory controller fetches stereo image data from the external memory or camera source to the ECC in raster-scan fashion and stores the incoming pixels of the left stereo image in the input image buffer, while the produced edge image is stored in the edge image buffer. The DCU reads the edge image pixels, performs the disparity map computation, and stores the disparity pixels in line buffers (disparity image buffer). The

WEU reads pixels from the line buffers to validate the candidate window and extract window pixels from the edge image and the left stereo image. All valid windows are then fed to the CE for classification. The proposed hardware architecture (Figure 5-5) for depth and edge accelerated object detection consists of the four major hardware units, as well as a system controller that optimizes accesses to the external memory, controls I/O operation, and synchronizes the other major units. These major units are described in the following paragraphs.

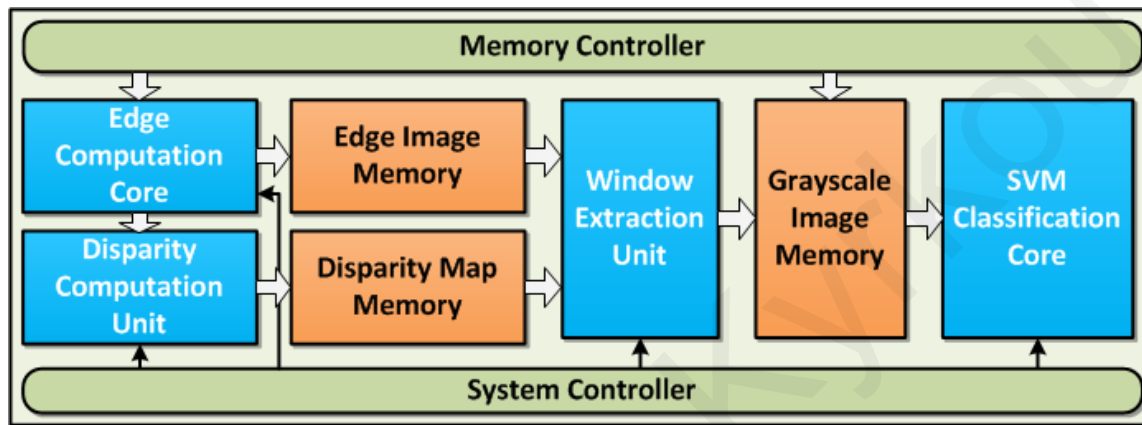


Figure 5-5. Depth and edge based system architecture

5.3.1 Edge Computation Core

The Edge Computation Core (ECC) integrated to the system implements a flexible and scalable Sobel edge detection architecture. While there exist several edge detection methods, this work integrates the Sobel edge detector in the FPGA-based detection system, mainly due to its simplicity and good performance on an FPGA [16]. The Sobel operator performs a 2D spatial gradient measurement on the input grayscale image using a pair of 3x3 convolution masks (Figure 5-6).

It employs hardware features such as parallelism and pipelining in an effort to parallelize the repetitive calculations involved in the Sobel operation, and uses optimized memory structures in order to reduce the memory reading redundancy. These features enable the architecture to obtain frame rates that exceed the 5,000 FPS for an image size of 320×240.

This is particularly important, as the overheads from the additional edge detection operations need to be small enough in order to obtain a speedup in the overall operation. The architecture of the ECC is shown in Figure 5-7 and consists of an I/O controller, a set of FIFO line buffers (scan-line buffers) used for temporary pixel storage and parallel window processing, and a series of convolution units (CONV), as well as comparators. The architecture is pipelined into 2 stages: *FETCH/WRITE BACK* and *PROCESSING*. In the *FETCH/WRITE_BACK* stage the I/O controller fetches pixel data from the input stereo image in a row-wise fashion and forwards them to the input port of the scan-line buffers. During the *PROCESSING* stage the scan-line buffers produce four successive 3×3 windows per cycle, which are convoluted with the 3×3 Sobel kernels by the convolution units. The masks for the Sobel Kernels hold data values between -2 and 2, as shown in Figure 5-6, thus the overall convolution can be implemented in hardware using shifters instead of multipliers. By avoiding the costly multiplication operation, higher frequencies are possible allowing integration of this method into object detection systems without affecting their performance, leaving the multiplier units to the more demanding classification engine. Furthermore, an approximation of the sobel detector was used which avoids the tedious tasks of square root and square operations as shown in Figure 5-6. The results of each CONV unit are compared against a threshold and produce 1 indicating the presence of an edge or 0 indicating its absence. The 1-bit pixel intensity values are concatenated into a 4-bit value which is stored in the edge map memory.

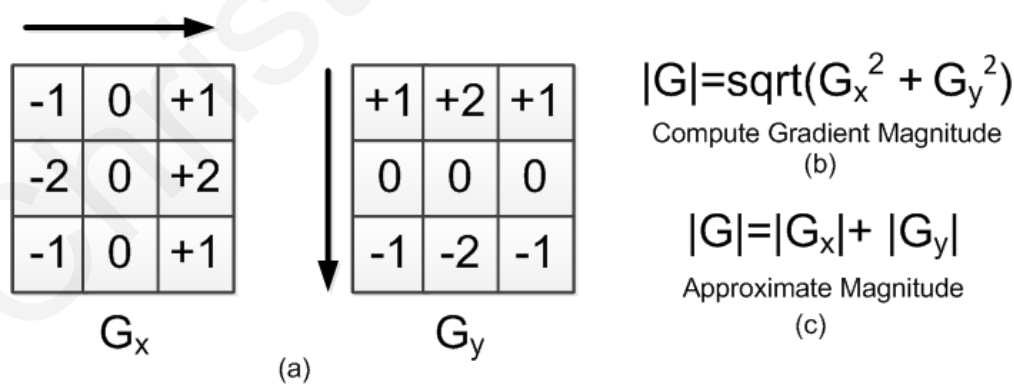


Figure 5-6. Sobel edge operator

(a) The two Sobel gradient masks in the vertical (G_y) and horizontal (G_x) edges can be applied vertically to the input image. (b) The results of the two are combined to find the magnitude of the gradient. (c) This process can be approximated by summing the absolute value of the two mask results.

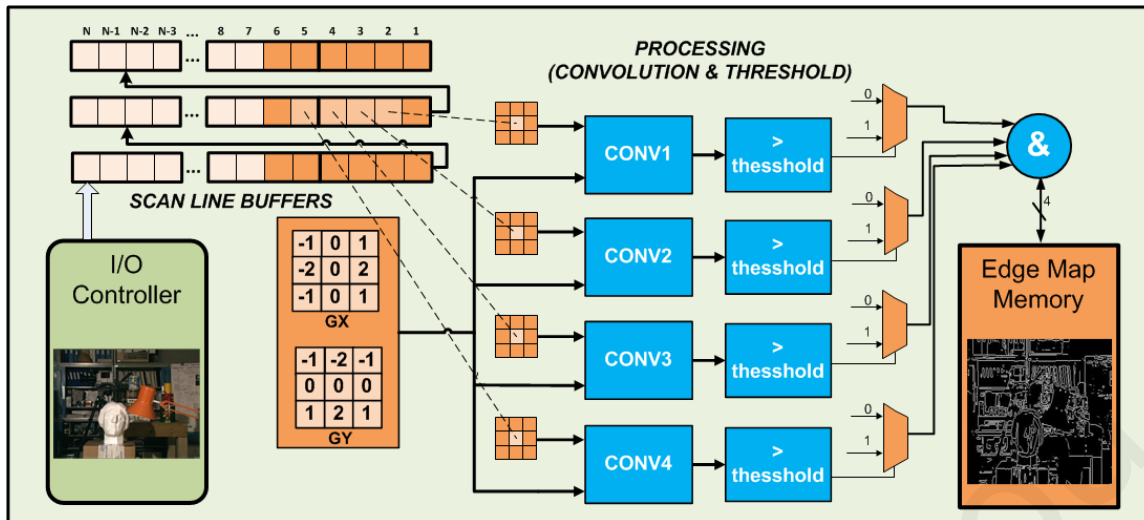


Figure 5-7. Edge computation core

The edge map of the left stereo image, stored in the edge image buffer, is used by the WEU, in the latter stages, to determine if a candidate window should be rejected prior to classification or not. Additionally, the produced edge-detection images are also used for the fast computation of the disparity map, as described in [170]. Thus the architecture incorporates two edge-detection units, one for the left and one for the right input image.

5.3.2 Disparity Computation Unit

The Disparity Computation Unit (DCU) used in the architecture extracts depth information from a stereo vision system, and its architecture is based on an improved version of the hardware architecture that was proposed in [170]. The architecture, which is shown in Figure 5-8, combines the sum-of-absolute-difference (SAD) matching algorithm with edge features for faster and more efficient processing. Specifically, the use of edge features reduces the hardware demands, since processing happens with one-bit pixels rather than eight-bit for grayscale images. Thus the area saved is used to increase parallelism and performance. This is significant for the efficient integration of this unit into an object detection system, since the overhead introduced must not affect the performance of the classification process.

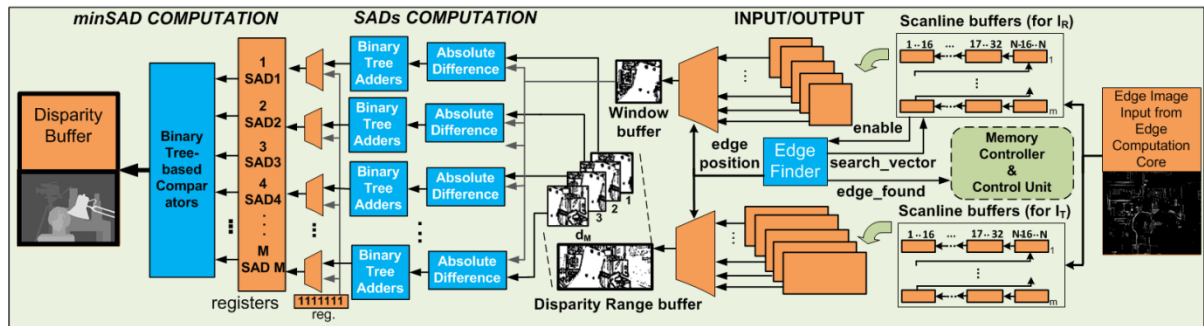


Figure 5-8. Disparity computation unit

The DCU follows the ECC in the computation flow, as it receives edge pixels from the ECC and uses them to perform correlation on the input stereo images to produce the disparity map. The DCU can process images with a disparity range of 80 pixels and window sizes up to 11×11 pixels. The DCU architecture consists of scan-line buffers in order to receive edge pixels from the ECC in streaming fashion. The scan-line buffers temporarily store the pixels and allow for parallel processing of many windows. As such, multiple parallel adders and subtractors are utilized which facilitate in the parallel implementation of the SAD algorithm. Through this parallel structure, the DCU is capable of producing disparity values every cycle. The disparity values are produced faster than they are consumed by the WEU and CE, thus, it is only necessary to store a couple of lines of the disparity map in the disparity image buffer. More disparity map pixels can be produced very rapidly, so there will always be data available for the WEU which follows the DCU in the computation flow. There are additional advantages of activating the DCU only when disparity values are needed by the WEU. First, energy is saved, since the DCU is not active for a large amount of time unless a lot of disparity values are not valid, in which case, the DCU will constantly produce disparity values. However, in that case, the CE will not be active again, resulting in energy savings. Second, on-chip memory requirements for storing the disparity map can be reduced and the remaining memory resources can be used to store the input image. The reader is referred to [170] and [171] for a more detailed description of the concepts used in the design of the DCU.

5.3.3 Window Extraction Unit

The Window Extraction Unit (WEU), shown in Figure 5-9, is the third unit in the detection system pipeline and performs the necessary operations for the validation of candidate windows, which are the window-size estimation and the accumulation of edge pixels. Additionally, it also performs the reverse mapping process that rescales large windows to the appropriate window size for the classifier ($m \times n$, can be either square or rectangular). The WEU is enabled once the DCU starts generating and storing disparity values in the disparity image buffer. The disparity map is sampled by the WEU every few pixels, depending on the object size, and thus not all values of the disparity map are processed. The WEU loads them from the buffer and proceeds to determine the corresponding window size of each disparity value using Equation 5-2.

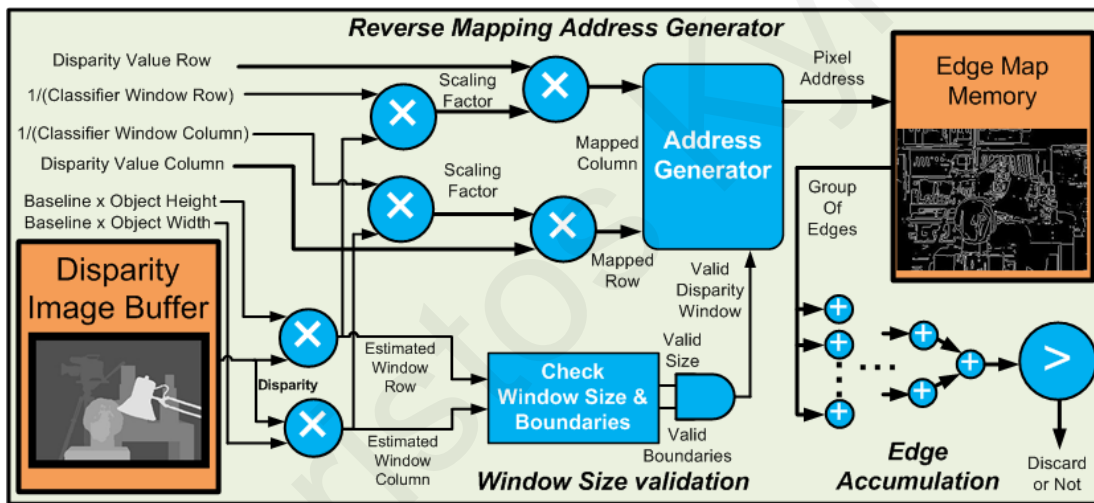


Figure 5-9. Window extraction unit

To implement the equation, the incoming disparity value is multiplied by the fixed-point preloaded value (O_{size}/b), producing the window size in pixels. The calculated window size goes through some comparators that check if the size is equal or larger than the classifier window size, and if the window is within the image ranges. If the conditions are met, then the corresponding window is extracted from the edge image buffer, using the reverse mapping process, and the edge pixels in the window are counted. The reverse mapping technique is implemented by multiplying the actual window coordinates with a predetermined scaling

factor (*computed window size/classifier window size*). This ensures that for every window, regardless of the estimated window size, only $m \times n$ pixel values will be read and processed by the subsequent stages. Details on the reverse mapping process are given in Figure 5-10. Through reverse mapping the edge image pixel values, which are either 1 or 0, are accumulated as they are fetched to find the number of edge pixels in the window. This process is implemented through an adder tree of simple 1-bit adders. Storing and processing of the edge image pixels happens in groups of four bits to improve performance, and as such, the accumulation is done for four edge image pixels as well. Once all the pixels are accumulated, the sum is compared to a threshold to determine if it should be discarded or not. If the threshold is exceeded, the WEU starts fetching window pixels, this time from the input image buffer using the reverse mapping technique, to the classifier. This requires first finding the window size and then fetching the pixels from that window that correspond to the equivalent classifier window size window.

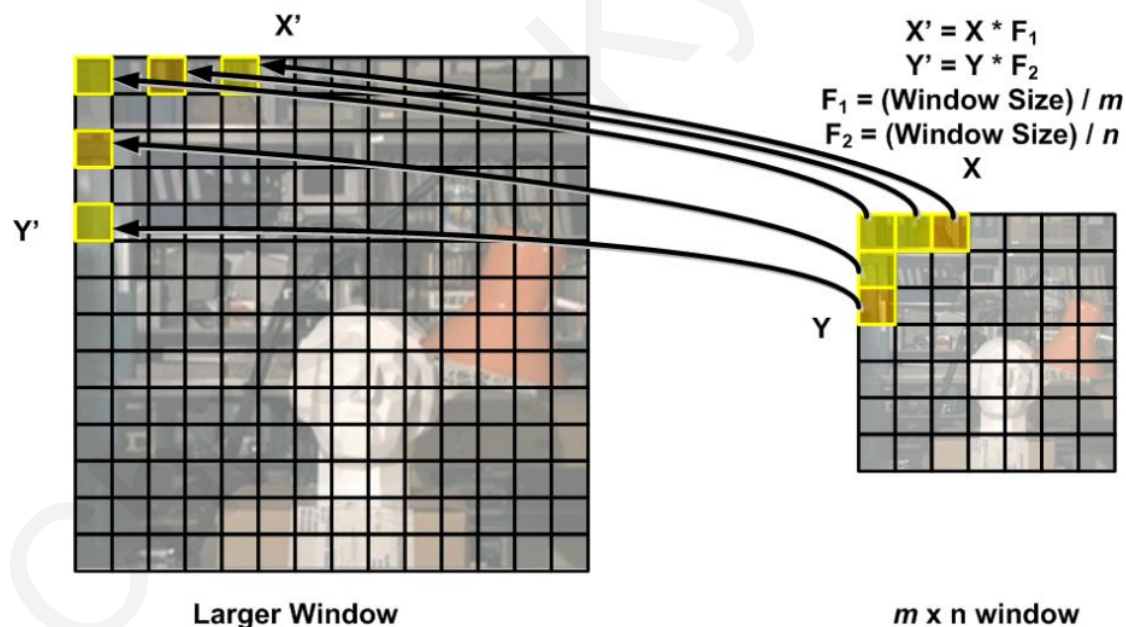


Figure 5-10. Dynamic image downscaling through reverse mapping.

The reverse mapping process involves reading only the pixels that correspond to an $m \times n$ window on a larger image. Each pixel address in the $m \times n$ window is transformed into an address corresponding to the pixel's location in the larger image using scaling factors F_1 , F_2 , which are computed by the larger window dimensions divided by the classifier window dimensions.

5.3.4 Classification Engine

The CE architecture is based on the SVM array processing architecture presented in Chapter 4. It consists of an array of processing elements (PEs) and reflects the processing requirements of the SVM computation flow, which requires the calculation of a kernel function that has both vector and scalar operations. The architecture consists of two types of PEs which perform the vector and scalar operations of the SVM computation flow (i.e., the vector and scalar units). The kernel used for the SVM is the second degree homogeneous polynomial $((x_i \cdot x_j)^2$ which has been extensively used for object detection [76],[136] and could be adapted to have less support vectors without sufficient accuracy loss [51] as shown in Chapter 2.5.3. As a result the vector units in the SVM architecture compute the dot-product between vectors while the scalar units perform the squaring of the dot-product result and the latter SVM classification flow operations (Alpha multiplication, accumulation, Bias processing).

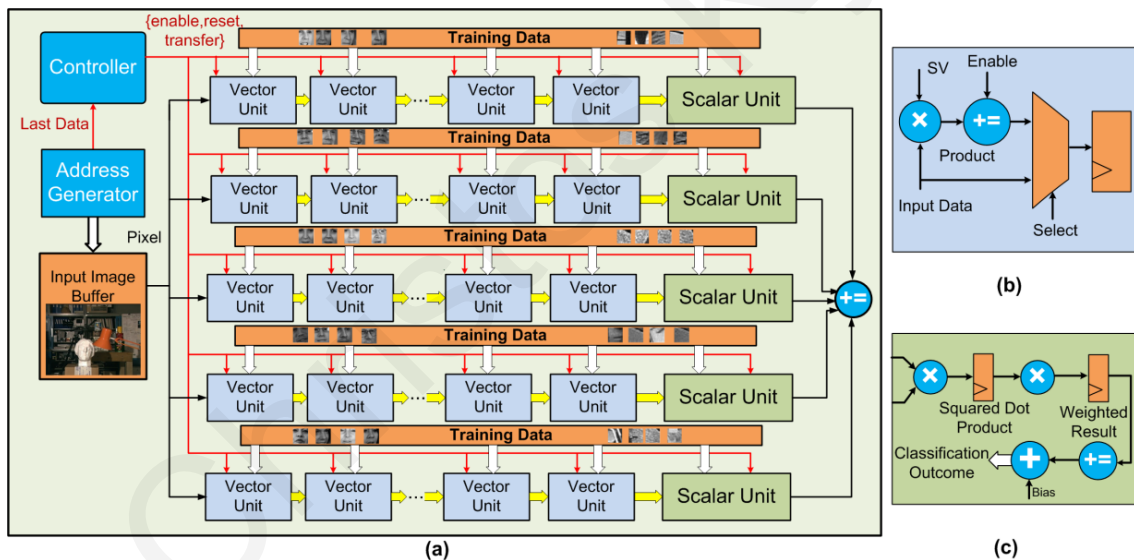


Figure 5-11. Support vector machine classifier architecture

(a) Support vector machine classification core. (b) Vector unit. (c) Scalar unit.

The architecture described in the previous chapter permits the SVM processing core to handle the processing of multiple windows, as each row can perform classification of one window. Additionally, it allows trading off processing the training data in parallel or

processing input windows in parallel. The latter, however, depends on the memory I/O and parallel access capabilities. The multi-window processing approach is useful when many windows are generated from a single image and need to be processed. Through the depth and edge directed search space reductions the number of generated windows is reduced, hence more important to classify a single window really fast. As such, the array-based architecture has been optimized for the specific task at hand. Specifically, in the case of the developed system dual port on-chip memories are used, which are available on the FPGA, for storing the input image, with one port used for writing while the other is used for reading, only one window can be accessed at a time. As such, the SVM classifier architecture was adapted in order to speed up the classification of a single window by processing as many support vectors as possible. The modified array architecture, which is shown in Figure 5-11, consists of five rows, each processing the same window but with different reduced-set vectors (RSVs) (training data), thus reducing the classification time. The results of each row's vector units are accumulated by that row's scalar unit. In turn, the result of each scalar unit is also accumulated to produce the final classification result. Using this approach, a window with corresponding dimensionality k_{dim} can be classified in $k_{dim} + (2 \times number_of_columns)$ cycles. The classification process is the bottleneck in the overall computation, since the ECC and DCU can produce results almost each cycle. Hence, employing a cascade classification structure such as the one proposed in Chapter 4 can lead to further performance improvements.

5.4 Experimental Methodology and Results

5.4.1 Experimental FPGA Platform and Methodology

The proposed architecture was evaluated after implementation on a Xilinx ML505 board (Virtex 5-LX110T FPGA), targeting the application of face detection as benchmark. The evaluation image dataset consisted of real-world images that were taken from a custom-built stereo vision system¹ as well as synthetic digital images. The stereo image capturing was comprised of two video cameras system (Figure 5-12) separated by a baseline distance of

¹ Two sony handycam digital HD video cameras - HD CX115E

77 mm, both with a focal length of 25 mm. Offline calibration was performed with a calibration pattern using [172]. Stereo image pairs were loaded to the FPGA board DRAM from the compact flash through a Microblaze subsystem that was used for initialization and verification purposes. After the initialization phase, the object-detection architecture was enabled, and data was retrieved from the DRAM by a custom memory controller. A total of twenty images per size (320×240 , 640×480 , and 800×600) were used to evaluate performance metrics, such as frame-rate and detection accuracy. Classification was performed by a support vector machine which was trained on 20×20 images using the 2nd degree polynomial kernel which was found to perform well for image-processing applications [76]. The SVM model was trained using MATLAB and the CBCL Face Database #1 [139] which was enhanced with additional face and non-face images from the world-wide-web. The training followed the guidelines from [51], [76] and the resulting SVM model was reduced to of 80 RSVs following the methodology proposed by Burges [19]. The parameters of the depth and edge accelerated object detection system as well as the SVM model parameters are summarized in Table 5-2.



Figure 5-12. Stereo camera system

TABLE 5-2: SYSTEM PARAMETERS

SVM Kernel	2 nd degree homogeneous polynomial ($x \cdot y$) ²	Number of Reduced Set Vectors	80
Reduced Set vector Encoding	8-bits	Alpha Coefficients Encoding	10-bits
Disparity Window	11 × 11	Disparity Range	80 pixels
Minimum Number of Edge Pixels Threshold	50 pixels	Edge Detection Threshold	175 pixels
SVM Classification Window Size	20 × 20	Disparity Search Overlap	5 pixels

5.4.2 FPGA Implementation Discussion

The system architecture was implemented as a proof of concept on a Virtex 5 LX110T ML505 FPGA board and was optimized for the specific FPGA hardware features and resources in order to be able to process images up to 800 × 600 pixels. Specifically, the image buffers are implemented as dual-port block RAMS, which are available on the FPGA, to facilitate the streaming nature of the operations. One port is used for writing image data and the other for reading. A total memory space of 800 × 240 pixels was allocated for buffering the input grayscale image. As such, a whole image of 320 × 240 can fit on the FPGA, for larger images (640 × 480 and 800 × 600), only a part of the image is stored on-chip. This is sufficient, as in most cases, the window will be available on the on-chip memory, and as a result, external memory access will not be needed. Furthermore, whenever the window pixels are not on-chip, this will indicate the presence of a very large object, which covers most of the image, as such, a large portion of the image will not need to be processed and so the overhead from accessing the external memory is negligible. The Sobel convolution process of the ECC unit was implemented using shift registers rather than multiplications to save hardware resources and increase frequency. The SVM array was comprised of a total of 80 vector units (5 rows and 16 columns), with each vector unit handling the processing of the input window with one reduced-set vector. A total of five scalar units was required for the implementation of the SVM computation flow. All the units in the SVM processing core require multiplication

and accumulation units, however, since the scalar units have a higher demand in bitwidth requirements, they were mapped on the DSP48E units of the FPGA for performance improvement, while the vector units were mapped on the LUT logic.

The FPGA resource utilization of the proposed system is illustrated in Table 5-3, which also shows relevant results from related works. The FPGA technology used plays an important role in the efficiency and performance of a design. A feature-rich FPGA platform that provides more processing power in the form of more reconfigurable logic, embedded multipliers, and embedded block RAM to further exploit parallelism, could potentially provide higher performance. However, the architecture design has to be scalable and utilize the FPGA resources efficiently in order to deliver the full performance potential of a given FPGA platform. As such, comparing architectures implemented using different FPGA technologies, even if it may be indicative of potential performance and required hardware, may not be fair. Thus, comparison is focused with works that have used the Virtex 5 FPGA technology, however, other works are also included in Table 5-3 for a more comprehensive view.

The reported figures for the implementation are for a system capable of processing images of up to 800×600 . Fewer resources would be needed if the system is only going to process lower-resolution images. The presented architecture requires less LUT resources than other works and only half of the available register resources of the targeted FPGA for the implementation of the architecture. The CE requires the majority of the DSP48E units, which could be reduced by reducing the number of rows in the classifier, possibly reducing, however, the classification performance per window. Architectures that use the Viola-Jones detection algorithm have an additional advantage in that they do not require many multiplier units, since the algorithm is mostly implemented with adders/subtractors and accumulators. On-chip storage is critical for reducing external memory I/O and increasing performance, as also demonstrated in [7]. Hence, the developed system utilizes the majority of available FPGA block RAMs to reduce external I/O. The targeted frequency was set to be the one offered on the available FPGA board, and as such, the critical paths of the design were pipelined to achieve a frequency of 100 MHz. The frequency can be improved with further optimization for higher throughput. With this frequency, the system manages to offer very good performance results.

TABLE 5-3: FPGA RELATED WORK SYNTHESIS RESULTS

Work	FPGA Platform		Logic Elements		Embedded Multipliers/ DSP48E	Block RAM / Embedded Memory	Operating Frequency (MHz)
			Slice Registers	Slice LUTs			
He [7]	Xilinx FX130T Virtex 5 ML510 board		37828 (46%)	67704 (83%)	161 (50%)	276 (93%)	73
Han [166]	Virtex 5 LX330		N/P	n/a	N/P	N/P	54
Sadri [72]	XC2VP20		N/P	17000 (74%)	N/P	N/P	200
Ming [153]	Altera Cyclone II EPC2C70		9679		N/P	856605 bits	100
Cho [84]	Virtex-5 LX155		21902 (22%)	84232 (86%)	7 (5%)	97 (50%)	N/A
Hiro moto [83]	Virtex 5 XCVLX330		55515 (26%)	63443 (30%)	N/P	N/P	160.9
Farrugia [167] ^a	Virtex 4 SX35		2466 (16%)		19 (9%)	8 (4%)	80
McCready [168]	Transmogri fier 2A Altera 10K100		31500		N/P	N/P	12.5
Presented Architecture ^b	Virtex5 LX110T ML505 Board	SVM Core	7128 (10%)	13555 (19%)	45 (70%)	46 (31%)	100
		Edge Unit	20868 (30%)	2812 (4%)	---	3 (2%)	
		Disparity Unit	1008 (1%)	31504 (45%)	---	---	
		Other	8438 (13%)	8316 (12%)	3 (5%)	87 (58%)	
		Overall	37442 (54%)	56020 (81%)	48 (75%)	136 (91%)	

^a Synthesis results for one PE on a Virtex4 SX35. Four PEs are instantiated on the specific FPGA.

^b Utilization when processing images up to 800 × 600. Less resources are needed for smaller images.

N/P - Not Provided | N/A - Not Applicable

5.4.3 Performance Results and Discussion

Performance in object-detection systems is measured in terms of frame-rate and detection accuracy. A real-time performance of around 30 frames per second (FPS) can be deemed sufficient for most video-processing applications, however, applications with multiple data streams and high-resolution video analysis may require higher frame rates. Both these metrics are affected by the number of windows that needs to be processed for each of the considered image resolutions. From the numbers illustrated in Figure 5-13, there is on average a 7,4 × reduction in the number of generated windows compared to the sliding window approach. Even in the worst case scenario where the depth and edge directed search corresponds to all windows being valid for classification, though highly unlikely, there is still a 2,5 × reduction

in the number of windows. Through this significant reduction the FPGA implementation of the proposed depth- and edge-directed object-detection system is capable of real-time performance for 320×240 and 640×480 images (271 and 42 FPS, respectively), while achieving near real-time performance of 23 FPS for 800×600 images which can be improved by adjusting the disparity map search granularity. The average frame rate achievable by the presented implementation as well as other systems is shown in Table 5-1. The bottleneck in the performance of the system is the CE, which takes a few hundred cycles to classify a window (depending on its size), whereas the ECC (because of the parallel window operations) and the DCU (because of the binary data processing) produce a result almost every cycle. Thus, they are capable of achieving over 100 FPS for all targeted image resolutions, as demonstrated in [171]. The achieved performance is then limited by the number of windows that need to be classified and the classification time per window. The work in [7] achieved 625 FPS for 640×480 images, however, the actual size of the processed image size is 80×60 , with only three object sizes considered. The FPGA system by Sadri et al. [72] can process 800×600 images in 90 FPS, however, this is achieved if only 25% of the image is searched through skin detection and the detection itself happens on binary images, and thus it is unclear how detection accuracy is affected.

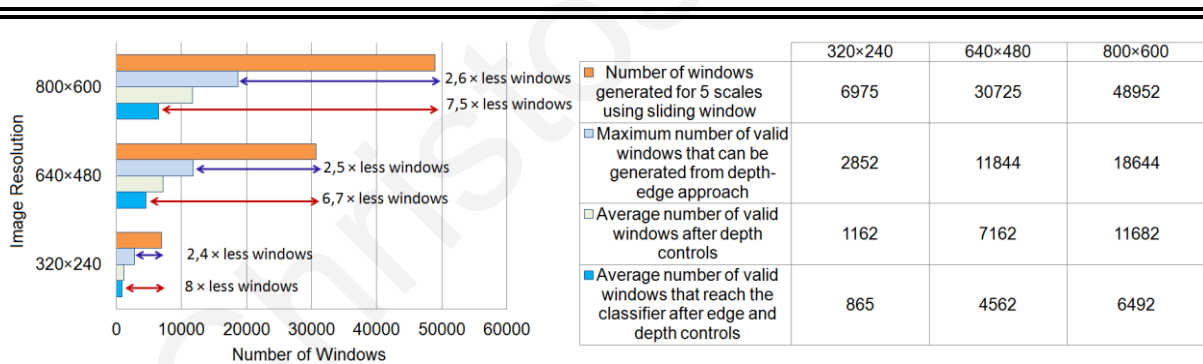


Figure 5-13. Reduction in the number of windows using depth and edge information

The depth of objects, the number of edge pixels in the image, and disparity map sampling step are all factors that affect the frame rate. Depending on the depth of objects and the number of edge pixels in the input stereo image, the frame rates may be a bit higher or lower. Coarser grain sampling of the disparity map could further improve performance, as fewer windows would be generated. An increase of the disparity search overlap for 800×600 images

from five pixels to ten pixels could potentially double the performance without any significant loss in detection accuracy. Increasing the disparity search overlap could enable the architecture to also process larger image sizes than 800×600 in real time. The proposed system architecture was tailored to the available FPGA resources, and as such, there are a few limitations that could be overcome by investigating an ASIC design especially for higher-resolution images. Some of the things that could be explored through an ASIC implementation are the design of on-chip memory structures that would allow for higher pixel throughput and optimizations on the CE architecture to increase classification performance per window.

Detection accuracy is an equally important performance metric for object-detection systems, and it heavily depends on the classification algorithm used as well as the training data. The Viola-Jones detection algorithm has demonstrated very good results in terms of detection accuracies for *2D* object detection, however, SVMs have also shown very good results, and literature suggests that the detection results are comparable [76],[103]. Furthermore, SVMs have also been used for *3D* object recognition [165], [121]. Detection results of the presented system and other related works are shown in Table 5-1. A direct comparison of the detection accuracies, however, is not fair, since the image datasets used in other works are comprised of a single image, and as such, they cannot be used to evaluate the stereo processing system. Additionally, the training data and preprocessing methods also impact the classification performance. The proposed depth-and edge-accelerated system can achieve classification rates of $\sim 80\%$ which would suffice for most applications. The detection accuracy is a bit lower than the Viola-Jones face-detection implementation in the OpenCV library [18], however, higher classification rates are possible by improving the training dataset and by incorporating other features. An inherent advantage of the proposed depth and edge search-reduction approach comes from the fact that the number of windows that are processed is reduced, as a result, the false alarm rate also decreases when compared to the traditional sliding-window approach. This is true regardless of which classifier is used, granted that it has acceptable discrimination capabilities. In the performed experiments the number of falsely classified faces (false positives) decreased an average of 52%, compared to the sliding-window approach, while the percentage of correctly classified faces (true positives) remained relatively the same. Some synthetic and real-world images used for the experiments and the resulting detections are illustrated in Figure 5-14.



Figure 5-14. Depth and edge directed face detection results

(a) Detection results using depth and edge information. (b) Disparity maps of input stereo images. (c) Edge detection results for the input images.

The proposed hardware architecture has been optimized for face detection, however, with minor modifications, it could be adjusted to perform detection of any object. First, the changes necessary in order for the architecture to be used for other applications are outlined. In general, only the classifier needs hardware changes, if an SVM produces adequate results, then a similar architecture to the one presented in the preceding chapter can also be used, however, any other classifier could be incorporated in the architecture if necessary (e.g., the Viola-Jones-based classifier presented in Chapter 3). The other changes concern specific parameters that need to be adjusted in each hardware module and are necessary in order to facilitate the new object of interest. Specifically, the window size for each detection scenario is different, and as a result, the pre-computed WEU parameters are adjusted to the object size characteristics (window height and window width in Figure 5-9). The threshold for the minimum number of edge pixels in a window could also be adjusted, since the window is now different. Additionally, it is also possible to introduce an upper threshold on the number of edge pixels depending on the object of interest to eliminate noisy regions or cluttered background regions. Finally, any feature extraction algorithm (LBP, HOG, etc.) could be integrated by implementing an appropriate processing unit that would perform processing on the valid windows prior to the classification process.

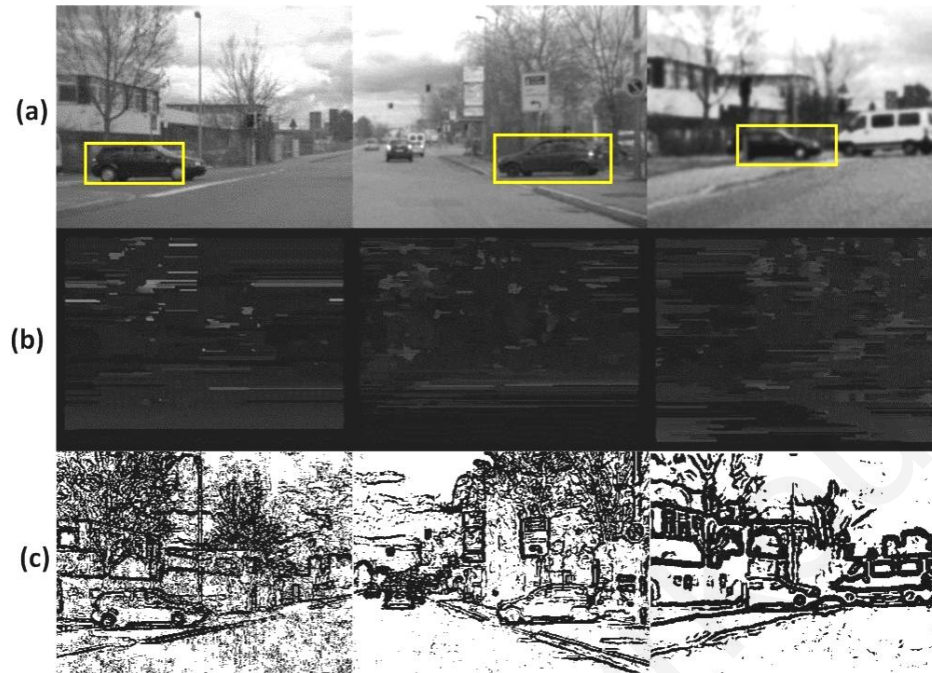


Figure 5-15. Depth and edge directed car detection results

Early simulation results for car side-view detection on 320×240 images. A polynomial SVM of 50 reduced-set vectors is used for classification, (a) Detection using depth and edge information, (b) disparity maps of input stereo images, (c) edge-detection results for the input images.

To illustrate that the proposed approaches could be used in other applications, simulations were carried out for car side-view detection for 320×240 cropped and resized test images from EISATS Set 1 [173] using a window size of 100×40 . Some detection results are shown in Figure 5-15. The necessary changes done to facilitate the car side-view detection are briefly described next. The disparity and edge-detection tasks were carried out with the same parameters as face detection, as they do not take any object-specific parameters, while the window-extraction process was adjusted for a 100×40 window (window size estimation parameters, scaling factors for reverse mapping, and number of edges threshold). The classifier architecture was adapted to the new training set. A 2nd degree polynomial SVM with 50 reduced-set vectors was utilized. Because of the larger search window size, it was possible to increase the disparity search overlap, which resulted in fewer generated windows compared to face detection. However, the time needed for classification and edge-pixel accumulation increased, as both tasks are dependent on the window size (vector dimensionality). The

hardware demands for the SVM classifier also changed because of the increased window size. The memory needed was increased, since the reduced-set vectors have higher dimensionality, however, the processing resources needed were reduced, as there were fewer reduced set vectors. Using the Xilinx synthesis and simulation tools, it was estimated that the expected performance for this application could reach up to 70 FPS for 320×240 images. The figure is lower than face detection, since the classification time per window, which is the bottleneck in the overall computation, was increased.

5.5 Conclusions

This chapter presented a search-reduction approach that utilizes depth and edge information and its hardware realization on an FPGA, which can be integrated into existing 3D vision systems in order to build efficient, embedded image-analysis systems. The implemented object-detection system offers improvements terms of frame rate as the number of windows that need to be classifier decreases significantly. Furthermore, the detection accuracy is also improved with respect to the sliding window approach, as a lot of windows are discarded, through the depth and edge related constraints, prior to the classification process which reduces the probability of erroneous classification.

CHAPTER 6

CONCLUSION

6.1 Overview and Concluding Remarks

Visual object detection is an important step for many embedded vision applications, such as human-computer interaction, surveillance, biomedical imaging, space missions, and automobile, that require computing systems with the ability to "see and understand" their environment and surroundings through visual means. A key challenge in integrating visual object detection in embedded applications is to meet real-time performance constraints. Traditional CPUs fail to take advantage of the inherent parallelism of such algorithms due to limited computing resources, and so cannot be used for embedded applications. On the other hand, the raw computing power of GPUs comes at a cost of higher power consumption which makes them unsuitable for embedded applications. FPGAs offer a reasonable trade-off between performance, power, and flexibility for the development of hardware architectures which can then be ported to ASIC for higher performance and improved power efficiency. Hence, this thesis has presented hardware architectures for the acceleration of two of the most widely used algorithms for object detection [3], the Viola and Jones detection framework, and Support Vector Machines (SVMs). Furthermore, it has also looked at how additional vision information in the form of 3D depth and edges can be used for further acceleration of object detection in hardware.

Overall, it has been demonstrated how utilizing array-based hardware architectures for the Viola and Jones detection framework and SVMs can offer the required real-time performance through the parallelization of the processing of many search windows, as well as a modular and scalable design. In addition, the hybrid architecture for cascade SVMs takes advantage of the cascade classification processing flow in order to offer real-time performance. It is also coupled with a hardware reduction method which enables scalable design and a meta-learning method for further improving performance. Finally, this research has also addressed the design of a hardware acceleration that utilizes depth and edge information in order to achieve real-

time performance by reducing the number of windows generated from the input image, thus proposing an effective alternative for the commonly used sliding window search approach.

As a whole the research conducted in this thesis aims at the design and development of novel and generic hardware architectures and design frameworks that will result in more efficient and real-time intelligent embedded systems which can be used in a wide range of applications.

6.2 Future Research Directives

6.2.1 Short Term Research - Pre Processing/Post Processing & Potential Improvements

There are a number of directions that future research can take in order to improve upon the findings of this research thesis. Some of the most interesting steps are mentioned next.

A. Integrating Features into the Array Processing Architecture for Monolithic SVM

Good quality features are critical towards achieving robust detection performance. As such, the first possible improvement is to integrate preprocessing and feature extraction capabilities into the SVM classification array presented in Section 4.2 in order to improve the detection accuracy by preprocessing each window prior to classification to compensate for illumination variations and noise. These features can either be Haar features [174] or LBP features [70] which are computationally efficient or preprocessing in the form of histogram equalization which is also a commonly used approach [76]. However, this needs to be done in an efficient and rapid manner and with a compact architecture in order to facilitate real-time performance and low resource utilization. Hence, such architecture need to either exhibit low latency or have a way to overlap the preprocessing delays with other tasks so that the overall performance will not be affected.

Such preprocessing/feature extraction algorithms can be integrated with the array architecture through the scan-line register array buffers (Figure 6-1). There are two types of feature extraction processes that can be integrated. First, there are histogram-based approaches that attempt to normalize the image using statistical information (e.g. histogram), or perform some form of illumination compensation. These are global methods and thus need to read the

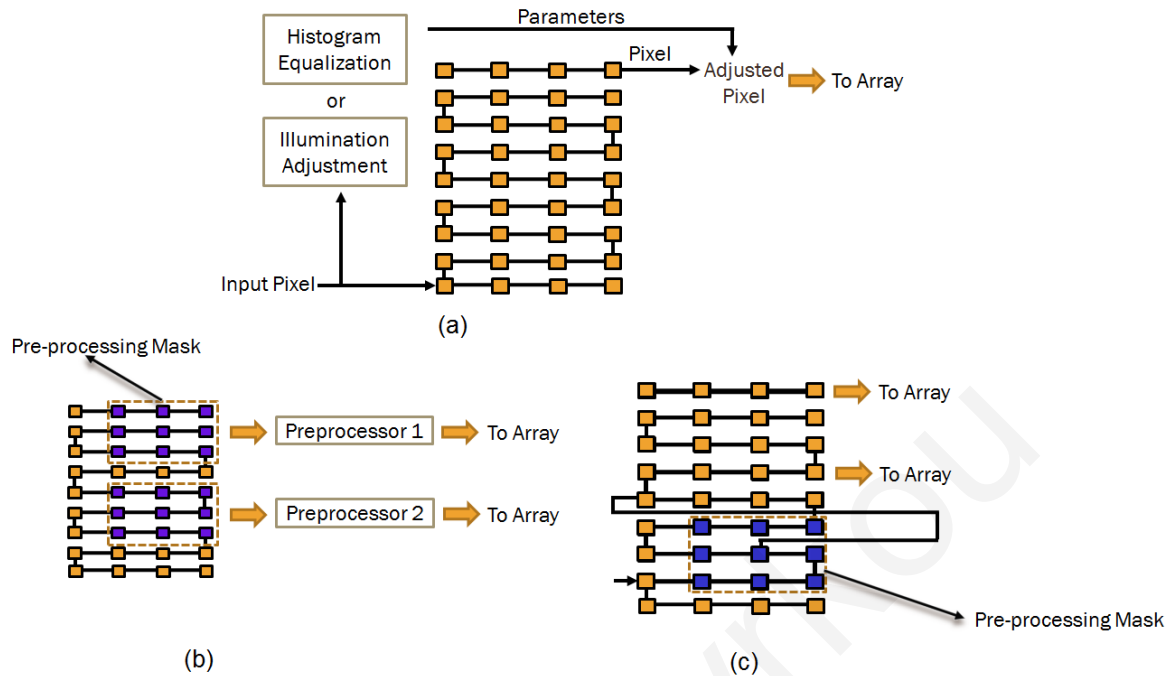


Figure 6-1. Integrating Preprocessing to SVM classification architecture

There are three ways to introduce preprocessing into the SVM array architectures. (a) Each pixel enters the scanline buffer sequentially and is also processed by the respective hardware for histogram generation or illumination correction. Once a whole window is loaded and the first pixel will be outputted to the array the adjustment parameters will also be ready for processing. (b) Each window is processed independently by the mask. The input pixels are loaded into the array and the convolution mask results (center location in the preprocessing mask processing array) are outputted to the array. (c) Processing of the mask happens for all windows with the same hardware. The results are propagated to the rest of the scan-line buffer and subsequently the array.

whole window in order to generate the necessary preprocessing information that will use to adjust pixel values. Such approaches can be integrated through a coprocessor to process pixel values while they are loaded into the scanline buffers. Once, all window values are loaded into the scanline buffer and are ready to be outputted, the histogram or adjustment parameters will also be ready and hence each pixel value that is loaded out of the scanline buffers and into the array is first adjusted based on the histogram parameters. The second approach convolves the image using a predetermined mask or operations in order to generate features or enhance the image quality. There are two ways to integrate such processes into the processing array, and in both cases specific registers are selected to act as input terminals to a preprocessing co-

processor where the convolution operation will take place. The first way is to process all windows from the same region in the scanline buffer and propagate the results to the rest of the scanline buffers. The second is to process the output of each window separately and provide the results to the array, while only input pixels will be propagated within the scanline buffer.

B. Online Training for Object Detection Systems

Learning involves changing the behavior of a model in a way that makes it perform better in the future. Thus enhancing the presented classification architectures/algorithms with the learning capabilities in order to retrain/adjust their parameters in the "field" opens up new possibilities and enhances the adaptability and flexibility of object detection systems to be used in dynamic environments and situations. In addition, to being able to improve what has been learned at run-time, adaptive training can also allow for a vision system to learn to detect new objects. However, there are two main questions that will need to be answered in order to achieve this: when will this retraining take place, and how will it be implemented. There are different ways that this can be done depending on the classification scenario.

First, a monolithic classifier is considered which for example can be the monolithic SVM classifier from Chapter 4. In general since machine learning approaches do not guarantee 100% correct detection there is no widely accepted method of knowing whether a classification is correct or not until it is examined by a human expert. Hence, without external supervision the machine learning approach will not be able to reliably readjust its parameters. Hence, in this scenario the retraining process must be done offline. To do this the system will first need to capture samples which are deemed as difficult, i.e. for which the classification result is close to the boundary (for class labels 1 & -1 it is around 0). Then a human expert will need to intervene in order to correctly label the stored data so that the system can be retrained using those samples. This process, given that there are thousands of samples that are classified at run time per image, can be time consuming if done regularly. In addition collecting samples that belong to the same scene/frame will result to samples that are similar without any variability and hence training will be biased. Considering these factors the capturing of these samples needs to happen every few frames and only keep a few samples per frame. In addition, implementing training algorithms on the same silicon as classification is

desirable in order to reduce external memory access and allow for a more autonomous system. Hence, there is a lot of potential in exploring a unified training/classification hardware architecture.

The second scenario concerns cascade classification systems where the key difference compared to the previous is that the succeeding stage can provide a classification label for the data to the previous stage although there is some probability of error as well. As such, every succeeding stage can act as a supervisor to the previous stage, and every previous stage can then be trained online whenever the classification results between the current stage and the supervisor are different. The training process can either involve a simple readjustment the classification threshold or more complicated methods where the classification model will need to be reevaluated. However, for the latter approach to be feasible the underlying machine learning algorithm needs to support on-line training (i.e. training using one sample at a time) such as neural networks. Hence, machine learning approaches such as the AdaBoost as well as Support Vector Machines which are trained with batch methods, that consider all training samples together, require research in different training methods and approaches that support on-line model evaluation.

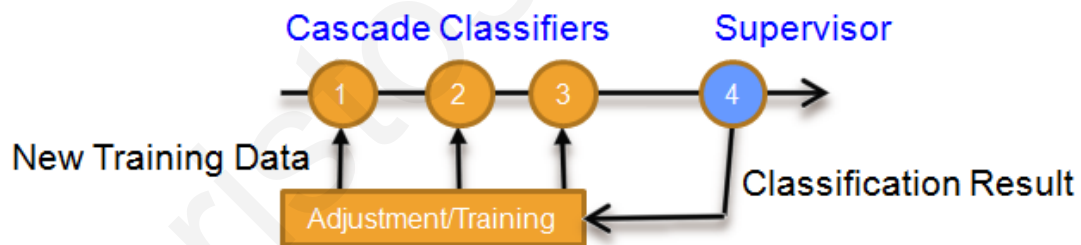


Figure 6-2. Online training of cascade classifiers

The last stage is used as a supervisor to the previous stages. The training data (weights)/threshold can be readjusted using the classification outcome of the succeeding stage.

C. Hybrid Processing Architecture for Cascade SVM

The presented architecture targeting cascade SVMs was optimized for the cascade classification flow. That is parallel vector element processing for linear SVMs and parallel SV processing for non-linear SVMs. The bottleneck in the classification phase was the slower sequential processing module which handled processing of multiple SVs a single element at a

time. However, additional modules can be integrated into the architecture that offer parallel processing of both support vectors as well as vector elements at different degrees in order to adapt to different demands and performance bottlenecks.

Another potential improvement would be to explore the possibilities offered by the neural-network response processing unit. As it is the unit is trained considering all responses collectively with a single neuron which makes the classification process simpler. However, better discrimination and classification results can be expected if pairs of classifier responses are considered, as this would increase the features and also provide more ways of exploiting the different sensitivities of each cascade classifier.

D. Depth and Edge Accelerated Object Detection

The presented alternative to the sliding window search based on depth and edge information provides an attractive framework on top of which the next generation of object detection systems can be developed. However, as the presented study was an initial proof of concept there are a number of ways that the overall framework can be improved. First, using a cascade classifier can dramatically improve performance. As such, exploring the 3D depth information processing with the AdaBoost approach seems to hold much promise. Second the depth computation process relies on edge features to compute the disparity and so the resulting disparity map is sparse. Essentially, this implies that some values in the disparity map may be erroneous. In order to account for these wrong values it is possible to compute more than one window size simultaneously based on slightly different scaling factors (e.g. $0.75 W_{size}$, W_{size} , and $1.25 W_{size}$) which also accounts for slightly different object sizes. Finally, the third way the framework can be improved is by altering the edge-based background rejection process which is carried out using a fixed predetermined threshold. The threshold in the presented experiments was determined empirically using a data set, and hence does not account for variations in lighting conditions. To account for this an adapting edge threshold can be used instead based on image information such as *mean* value of pixels and *variance*. Also the framework only accounts for uniform background regions but it is possible to enhanced it by adding an adaptive upper edge threshold, following the same methodology, in order to discard cluttered and noisy background regions as well.

E. Integration with Other Real-Time Vision Tasks

The real-time implementation of object detection systems can enable new applications and additional tasks that can be integrated either before or after the detection phase [1]. Typically, these processes are considered independently and not much research has been made to integrate them together in the same hardware architecture. There are many benefits stemming from such integration. By integrating motion estimation [175] for instance the search space involved for object detection will be reduced since it will be possible to predict the movement and next location of objects within a frame. Recognition and analysis tasks[1] can also be integrated as latter steps to enhance the potential of object detection systems. Analyzing the state of an object and recognizing its specific identity from objects of the same class can be considered a major step towards more intelligent, aware and autonomous embedded vision systems. In contrast to the object detection processes recognition and analysis algorithms operate on the detection results which is a substantially smaller portion of the image space. In addition they are also based on feature extraction and classification algorithms so in this respect similar hardware architectures and approaches can be used for implementation of recognition and analysis algorithms.

6.2.2 Long Term Research

A. Neuromorphic Visual Processing

The algorithms concerned in this thesis belong to the fields of computer vision and machine learning. These algorithms perform object detection through computational methods which do not exhibit similarities with the human visual detection system. However, understanding the fundamental mechanisms used in the visual cortex, can enable the design of new vision algorithms and hardware fabrics that mimic the visual cortex information processing [176],[177]. This paradigm shift is possible through the insights provided by neuroscience in the past few years [177], and can be used to design the next generation of vision systems that exhibit improved power, faster processing speed, flexibility and higher detection/recognition accuracies relative to existing computer vision systems [176]. The advantage of neuromorphic vision systems stems from the fact that they are stimulus-driven meaning that they are only triggered by significant events which increases their efficiency in term of computations,

storage and power. Researchers can engineer these principles into bio-inspired embedded vision technologies in two ways. The first is to develop basic processing blocks based on spiking neural network CMOS architectures which when connected together will be able to mimic the structure and operations of the visual cortex [176], [178]. The second, is through understanding of the computational principles and information processing of biological systems which will allow the development of computational methods/algorithms that can perform equivalent tasks akin to the biological ones, in order to improve the efficiency of traditional computer vision methods. One example is saliency estimation [179] which attempts to determine visually important parts in an image in order to focus the computation on those regions. Another is the use of multimodal image processing [180] in order to improve efficiency by extracting relevant features and reducing redundancy. It is anticipated that bio-inspired neuromorphic vision have the potential to outperform conventional frame-based vision approaches and thus can be used to realize advanced functionality like 3D vision, visual feedback loops, and others, in real-time.

B. Embedding Intelligence in Computing Systems

Pattern recognition and machine learning are important towards embedding intelligence into computing systems for a wide range of applications. The research in the thesis was concerned with classification and pattern recognition algorithms targeting visual object detection systems. However, the presented architectures, design approaches, and algorithms targeting object detection are generic enough so that they can be used for other real-time embedded classification applications that are in need for acceleration through hardware platforms, and exhibit properties such as regular data access and streaming input data. As such the applications that can be targeted by the presented architectures include but are not limited to financial prediction applications [181], speech recognition [40], handwritten digit recognition [40], communication channel equalization [182] and fault detection in various systems [183], as well as other applications from aerospace, and communication to networking and broadcasting [184]. Finally, pattern recognition and classification algorithms cannot only be used to design intelligent systems but also for the monitoring of intelligent systems as well. interestingly, existing approaches can be used to monitor processing systems found on-chip [185] in order to determine abnormal system behavior and patterns that indicate the presence

of a fault or to predict future events in order to avoid hazards and bottlenecks in processor performance. Overall, the presented architectures have the capability to be adapted to the needs of the application in terms of data storage and flow, as well as input/output rate requirements which enables them to be used in a variety of existing as well as emerging applications.

Christos Kyrkou

Christos Kyrkou

APPENDIX A:

A CASE STUDY FOR EDGE-ACCELERATED OBJECT DETECTION

Edge information can be used to discard non promising regions, as shown in Chapter 5, (image regions that do not exhibit high pixel changes and thus there is a high probability that these regions will not contain any useful information), eliminating them from the detection process. The edges are extracted from the image by using the Sobel operator, and are then used to identify background and non-background regions. In addition to background removal edges show the outline of an object, and so they can be used as indication of the size of the object as an alternative to sliding window and 3D information. Consequently, edges can be used to construct a window for the object. Each window can then be processed to determine if it belongs to the background or not. The classification algorithm then classifies only non-background regions. This appendix presents a hardware architecture that performs edge-directed object detection as a low-cost alternative to the traditional sliding window process.



Figure A-1. Edge-based window extraction method

(a) Input Image (b) Compute edge map from input image and find first edge (c) Find second edge and compute their distance (d) Use distance to form square window. Then compute candidate window mean and if it exceeds a certain threshold send it to classifier

The first task of the edge-directed object detection algorithm (Figure A-1) is to search for edges that may enclose an object and then use them to construct a candidate window. Working in a row-wise fashion, this edge search process locates an edge and uses the position of that edge as the top-left coordinate of the candidate search window. The algorithm then searches for the next edge point at the same row starting from the position that indicates the minimum window size until a certain pixel limit. That point is considered as the top-right coordinate of the candidate window. The coordinates of the top-left and top-right edge points are next used

to extract a candidate search window of square shape. The mean value of the window is computed next to identify whether it contains enough information to be considered a non-background region or not. If the candidate window does not exceed a certain threshold it is considered as background and skipped to extract the next candidate-window. Otherwise, the corners of the candidate window are used to extract that window from the original image, which is in turn given as input to the classification algorithm.

The proposed hardware architecture (Figure A-2) for edge based object detection consists of three major hardware units, the Edge Computation Core (ECC), the Window Extraction Unit (WEU) and the Classification Engine (CE). The system also consists of a system controller that optimizes accesses to the external memory, controls I/O operation, and synchronizes the other major units.

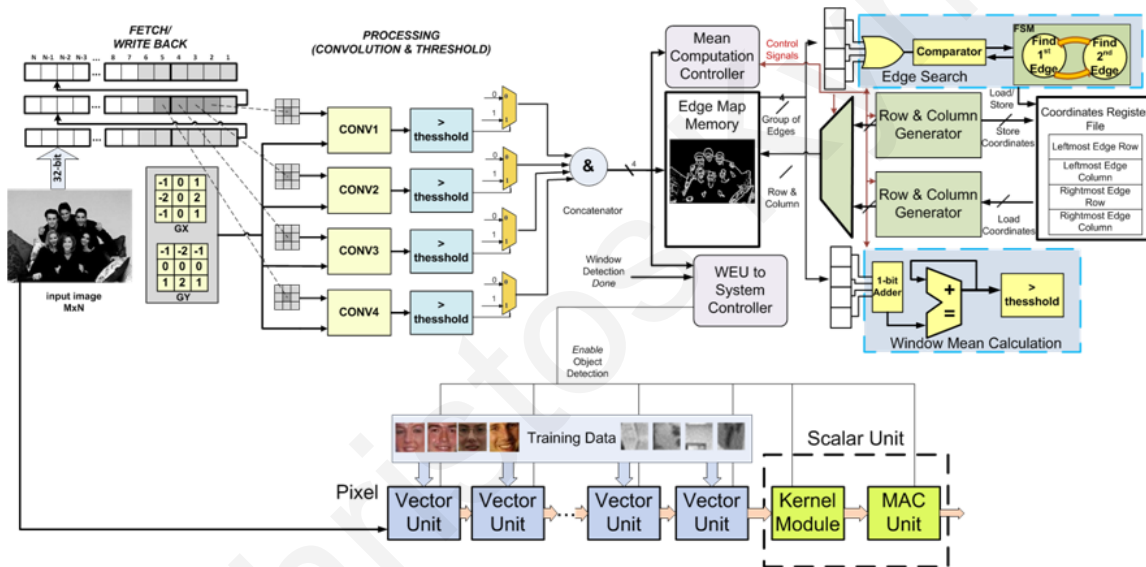


Figure A-2. Edge acceleration system architecture

The overall operation of the system involves edge computation, candidate window extraction, and window classification. The memory controller fetches data from the external memory to the ECC in raster-scan fashion, and stores the incoming pixels in the *input image buffer*. The ECC first computes the edge map of the first frame, which is stored in the *edge map memory*, and then the WEU starts the edge-search process, enabling the CE when a valid window is found. The ECC and CE are pipelined and thus operate concurrently. However, the

ECC only writes to memory regions that are already processed by the WEU to maintain data consistency. The system can have multiple CEs operating in parallel to increase performance, each controlled by a WEU that searched different rows in the edge map.

The Edge Computation Core (ECC) integrated to the system implements a flexible and scalable Sobel edge detection architecture. It employs hardware features such as parallelism and pipelining in an effort to speedup the Sobel operation, and uses optimized memory structures in order to reduce accesses to the input image memory. These features enable the architecture to obtain frame rates that exceed 5,000 FPS for an image size of 320×240 . This is particularly important, as the time allocated for edge detection must be small enough in order to obtain a speedup in the overall operation (edge detection + object detection).

The Window Extraction Unit (WEU) is responsible for searching for pairs of edges that are used to create a candidate window. The edge search procedure is carried over by examining groups of edges together, rather than focusing on single edge image pixels as in [158], in order to take advantage of the parallelism offered by the hardware and speed up the search procedure. The WEU is tightly coupled with the ECC and the *edge map memory*. The WEU reads groups of pixels from the *edge map memory* and passes them through an *OR* gate which if asserted will indicate the presence of an edge. Then the coordinates of the first pixel in the group are marked as the coordinates of the top-left-most pixel of the candidate window. If an edge is found in the following pixel groups in the same row the coordinates of the first pixel of that group are marked as the coordinates of the top-right-most pixel of the candidate window. The two marked coordinates are used to form a square candidate window. The candidate window belongs to the background region if the number of edges in the window is below a certain threshold. Otherwise, the WEU enables the CE and passes it the coordinates of that window to begin classification.

The classification is done using a simple, yet powerful Support Vector Machine (SVM) hardware architecture which is a variation of the processing array presented in Chapter 4.2. The proposed architecture consists of a chain of processing elements and reflects the processing requirements of the SVM computation flow which requires the calculation of a kernel function that has both vector and scalar operations. It consists of two types of processing elements, one type for the vector operations and one for the scalar operations.

Multiple vector units are pipelined to form a row and allow parallel processing of the training data, while a single scalar unit, which is the final pipeline stage, is shared amongst the vector units. The SVM architecture is modular and scalable, and thus allows for multiple rows to be instantiated all sharing the same training data, and each processing one input window.

The proposed architecture was implemented using a Xilinx ML505 board (Virtex-5 LX110T FPGA) as a proof of concept, targeting face detection. The Microblaze soft processor was used as the system I/O controller to handle tasks such as memory transfers and external I/O. Input images of 320×240 pixels were stored on the on-board DRAM, and were used as input data to the system. Visual output was directed to a digital monitor, through the on-board DVI output. The ECC only adds an overhead of just 2% compared to the sliding window system as shown in Table A-1. A test set of 20 320×240 images were used for the evaluation of the proposed architecture and detection results are shown in Figure A-3. The training of the Support Vector Machine classifier was carried out in MATLAB R2010b using the face detection database from [139] and the classifier is able to process windows of 19×19 resolution. Two different configurations were setup to evaluate the performance of the proposed architecture: one utilizing the edge information to find candidate windows, and one using the traditional sliding window approach.

The performance of edge accelerated object detection system is measured by the time necessary to compute the edge map of the input image and the time needed by the classification engine (in this case SVM) to classify the generated windows from the input image. However, since the ECC is capable of very high frame-rates (an order of magnitude greater than that of the CE) the bottleneck of the system is the CE. The performance speedups when using the edge-directed search and the traditional sliding window search are compared observing an average speed up of $\sim 5 \times$. This speedup is primarily attributed to the reduced number of windows that are generated and need to be classified. Overall, the sliding window process generates 4405 windows achieving 12 *FPS*, while the edge directed process classifies ~ 900 windows which results in an average of 60 *FPS*.

TABLE A-1: CLASSIFICATION SYSTEMS FPGA SYNTHESIS RESULTS

FPGA Resources	Logic Elements		Embedded Multipliers DSP48E (64)	Block RAMs (148)	Frequency (MHz)
	Slice LUTs (69120)	Slice Registers (69120)			
WEU	222 (< 1%)	98 (< 1%)	---	---	100
CE	14,385 (20%)	5,197 (8%)	8 (13%)	37 (24%)	
ECC	1,073 (2%)	371 (1%)	---	3 (2%)	
Microblaze System	7,016 (10%)	8,180 (11%)	3 (4%)	48 (33%)	
Sliding Window System	21,401 (30%)	13,377 (20)	11 (17%)	88 (59%)	
Edge Detection System	22,696 (32%)	13,857 (20%)	11 (17%)		



Figure A-3. Detection results using edge and compared to sliding window

Detection accuracy is an important performance metric for object detection systems, and is affected by the search method used. In the case of the proposed edge approach the number of falsely classified faces (false positives) decreased by 78% while the number of correctly classified faces (true positives) remained relatively the same with only 6% decrease. The reduced number of false positives is attributed to the reduced amount of data that are classified compared to the sliding window approach. The reduced true positive rate is due to the absence of edges around some objects which does not allow a window to be formed around those

objects, and also because there is no clear path between two edges to form a boundary of an object. The true positive rate can be improved by using other edge detection algorithms, or by searching for edges in neighboring rows as well. Another factor for discrepancies between the edge-directed search and sliding window search is that with the latter approach because of the gap between successive windows, a window may not be centered on the object and thus the object is not fully captured in the window. A way to avoid this is to reduce the window step size which, however, increases processing demands so it is a choice of trading-off accuracy and performance.

Experiments were also performed to compare the two configurations in terms of energy consumption using the Xilinx 12.4 X-Power Analyzer tool and image data as input to the system. Results indicate an approximately 73% energy reduction in energy per frame which is attributed to the lower number of generated windows.

The results indicate that the hardware overheads for the WEU and ECC are minimal (~2%), illustrating the potential of the proposed architecture to be used in embedded and mobile applications where hardware resources are limited, in order to accelerate the visual object detection process.

APPENDIX B:

IMPACT OF FEATURES ON SVM CLASSIFICATION

This appendix presents results concerning the impact of different features on the resulting accuracy using the very popular application of face detection [3] as the benchmark. The classifier used for evaluation is a 2nd degree polynomial SVM which has widely been used for object detection [76], A standard small-scale face database [139] was used for evaluation without any additional samples included or enhancements using bootstrapping. Furthermore, full frames were obtained from [147] to also test the results on unknown full 320 × 240 images. A description of each database is shown below. Finally, the search process involved searching 5 pyramid images and extracting a window every 5 pixels.

The presented results do not provide conclusive results and are not meant for absolute comparison but rather to demonstrate how features can impact the classification accuracy. Another factor that also needs to be considered, especially for embedded applications, is the computational complexity of each feature type. The reader is referred to Chapter 2.6.1 for a brief overview of each feature extraction algorithm.

- **Database:** CBCL Face Database #1 [139]
 - **Available Online:** <http://cbcl.mit.edu/software-datasets/FaceData2.html>
 - **Training Dataset:** 2429 Face Samples | 4548 Non-face Samples
 - **Training Test Set:** 472 Faces | 23573 Non-face Samples
 - Images scaled to 40×40 pixels from original 19×19 size

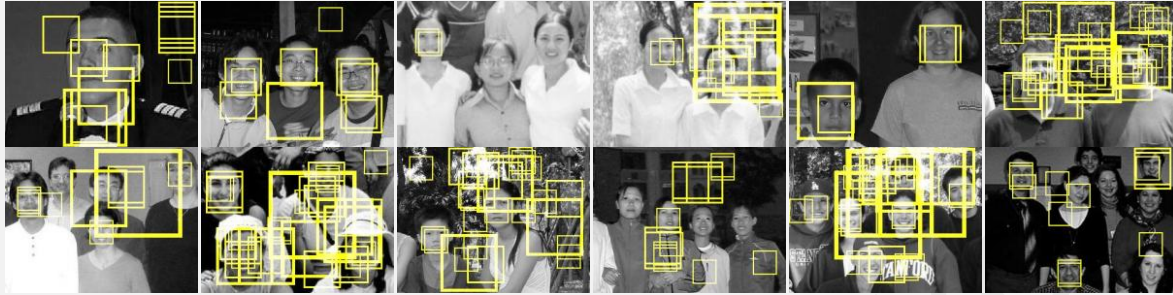
- **Full Frames Test Database:** Bao Face Database [147]
 - **Available Online:** <http://www.facedetection.com/facedetection/datasets.htm>
 - Images scaled, cropped and/or resized to fit 320×240 resolution

(1) Raw Pixel Values

Features	Original pixel values	Data scaling	0-1
SVM Kernel	Polynomial	Number of support vectors 454	Vector Dimensionality 400

CBCL Test Set Accuracy: 91.8%

Full Frame Results



Remarks: Using the raw pixel values [98] as input to classification requires no preprocessing. However, as is evident from the full frame results the number of false detections is very high. This happens because of the fact that the within class variability is very high, hence illumination changes and pixel intensity variations result in a high missclassification rate. The raw pixel values offer an attractive approach, however, the dataset needs to be enhanced significantly to achieve better discrimination.

(2) Histogram Equalization

Features	Normalized Pixel Values	Data scaling	0-1
SVM Kernel	Polynomial	Number of support vectors 361	Vector Dimensionality 400

CBCL Test Set Accuracy: 93.6%

Full Frame Results



Remarks: Histogram equalization [76] is a common preprocessing method for object detection however, it is a computationally intensive task as whole window needs to be processed prior to classification. Furthermore, the equalization process may sometimes distort the images resulting in missed detections.

(3) Edge Detection

Features	Edge Image Pixel Values	Data scaling	0-1
SVM Kernel	Polynomial	Number of support vectors	Vector Dimensionality
		1340	400

CBCL Test Set Accuracy: 86.6%

Full Frame Results



Remarks: The advantage of using edge features [72] is that the preprocessing is computationally efficient and also the classification is done on binary images instead of greyscale. However, the trade-off is that a lot of information is lost and so there is a high number of false detections. Instead the edge based classification can be used to filter out unnecessary information prior to the final more accurate classification.

(4) Histogram of Oriented Gradients

Features	Histogram Values	Data scaling	0-1
SVM Kernel	Polynomial	Number of support vectors	Vector Dimensionality
		244	128

CBCL Test Set Accuracy: 94.7%

Full Frame Results



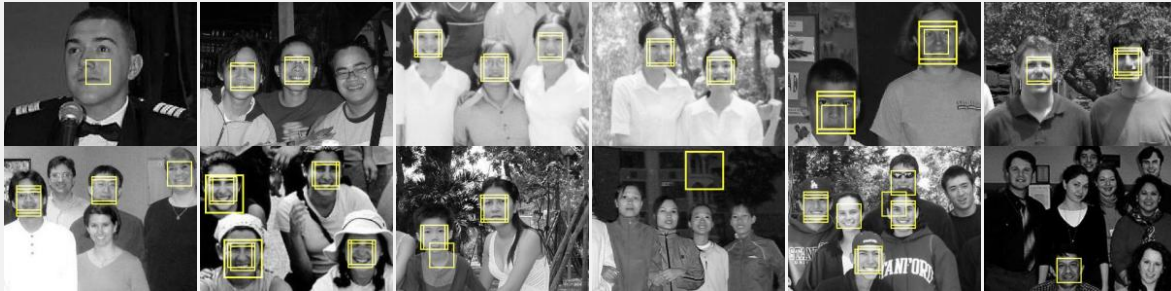
Remarks: The histogram of oriented gradient [61] features is a powerful tool that can capture shape characteristics of an object thus achieving high accuracy. However, there disadvantage is that they are computationally demanding and thus slower than the other approaches.

(5) Local Binary Patterns

Features	Histogram Values	Data scaling	0-1		
SVM Kernel	Polynomial	Number of support vectors	395	Vector Dimensionality	400

CBCL Test Set Accuracy: 95%

Full Frame Results



Remarks: Local binary patterns [70] analyse the relationships between neighbouring pixels and capture histogram information at three different levels. They offer accuracy comparable to histogram of oriented gradients but are computationally more efficient and thus may be preferred choice for embedded applications.

APPENDIX C:

A NOTE ON THE HARDWARE IMPLEMENTATION OF ARTIFICIAL NEURAL NETWORKS

A. *Artificial Neural Networks*

An artificial neural network (ANN) is an information processing computational system inspired by the basic properties of the biological nervous system of the brain [42]. The brain learns from experience and so ANNs can also be trained to classify input data into two or more categories depending on given training samples. An ANN consists of interconnected layers of processing elements called *neurons* (Figure C-1). The number of layers ranges from just one to several depending on the network structure. Direction and density of interconnections between layers and neurons may also vary, according to the application, depending on whether the network is feed-forward (data flow only moves forward) or feedback (output from a next layer neuron is an input to a previous layer neuron). Each of these network types can be fully or partially connected. Although there are useful networks which contain only one layer, or even one processing element (such as the compact neural network in Section 4.3), most applications require networks that contain at least the three normal types of layers.

The core processing element of an ANN, the neuron, can be mathematically modeled as an object with n input signals and one output [14]. Each neuron input is assigned a weight value during the training phase of the ANN. This weight determines the input's influence in the overall computation. In the neural network structure, neurons perform a multiplication of inputs with the respective neuron weights and then perform different operations such as accumulation, maximum, minimum or another operation (Figure C-1). Following the weight processing, a threshold value is subtracted from the result and it then goes through an activation function. Activation functions are used to describe the non-linear output behavior of a neuron and to saturate the output within a certain range. The output of the activation function

is then propagated to the next layer or the network's output. Some of the most popular activation functions that are used are given below (Table C-1).

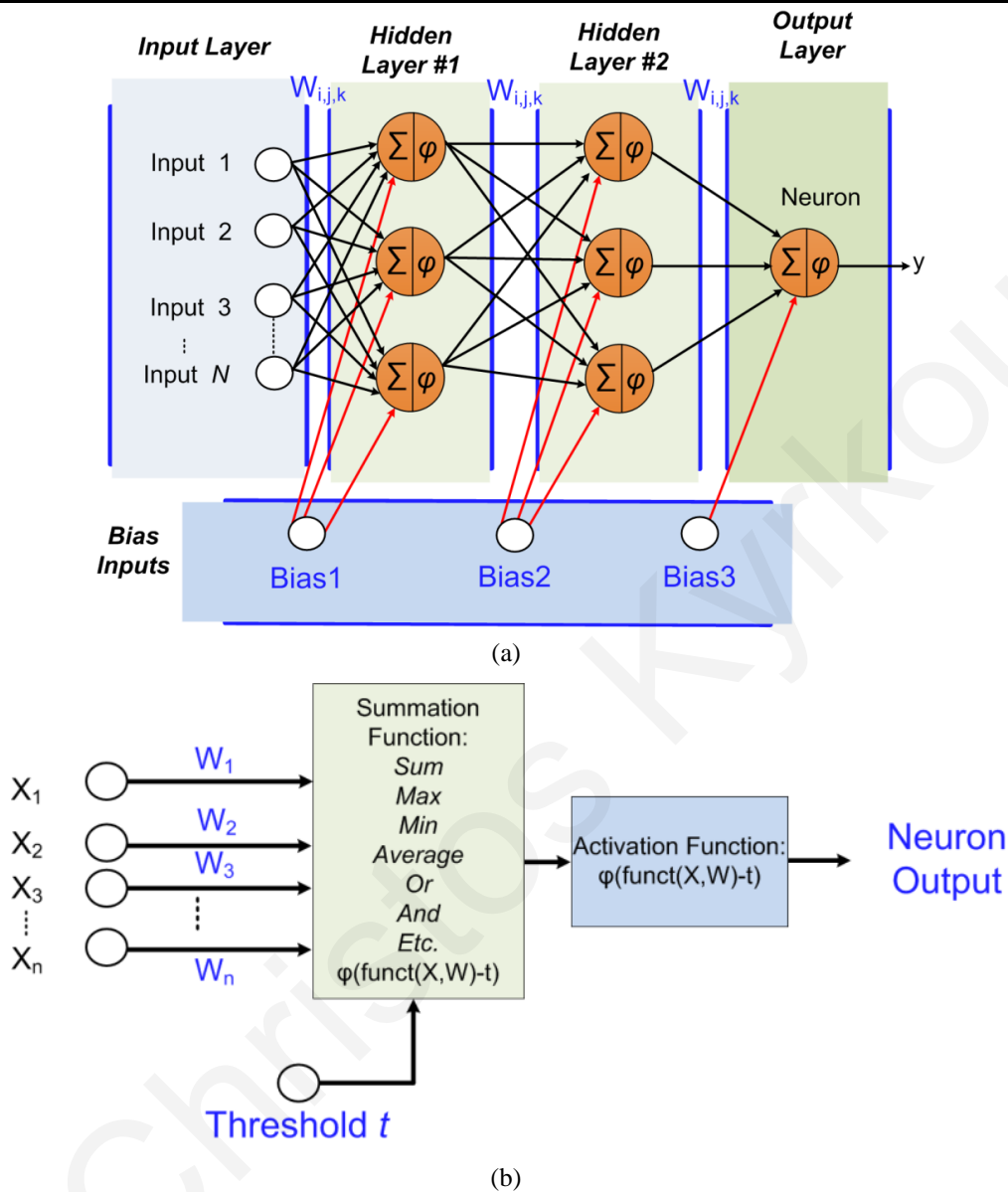


Figure C-1. Neural network and neuron model

(a) An example of a fully connected multilayer neural network (b) Model of a neuron, the basic processing element of a neural network

ANNs are typically trained using the backpropagation algorithm [17] either using the batch approach or online training. The weights and threshold values of the neural network are determined during this training phase which is typically performed off-line, using existing

software training algorithms. The classification operations is typically needed at run-time and needs to be performed in real-time. Hence, hardware implementations of neural networks have received noticeable interest [186]. The major trade-offs and challenges for the hardware implementation of neural networks are discussed next.

TABLE C-1 COMMON ANN ACTIVATION FUNCTIONS

Step Function: $\varphi(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$	Equation C-1
Hyperbolic Tangent: $\varphi(x) = \tanh(x)$	Equation C-2
Sigmoid: $\varphi(x) = \frac{1}{1 + e^{-x}}$	Equation C-3

B. Hardware Implementation Challenges

▪ Training Data Format and Representation

The ANN training data includes the neuron weights and threshold values. The weights are usually real numbers represented in floating point arithmetic. The most common representation of the training data for hardware implementation is fixed-point arithmetic. This involves associating a specified number of bits to represent the integer value and a specific number of bits for the decimal value. Such representation is easier to implement than floating-point because the multiplication operation is performed in the same way as with integers, thus no special floating-point hardware is necessary. The other issue is the precision of the representation, finding the optimal number of bits to use in the representation in order to minimize the memory requirements and preserve the targeted accuracy rates. The optimal number of bits required is usually found through high-level simulations and there have been previous attempts in optimal weight precision. All the aforementioned decisions not only affect the precision and operation of the ANN but also the memory requirements of the algorithm. The weights need to be stored either in an on-chip or on external memory. Ideally, the best scenario is to place them in on-chip memories, however, sometimes it can be necessary to trade-off accuracy for precision by limiting the number of bits used for representation to fit the weights on-chip which would significantly benefit the performance. Again, these trade-offs are explored through high level simulations prior to the actual implementation.

- *Algorithm-Specific Operations*

The main and most common operation performed by a neuron is the multiply-accumulate (MAC) operation and the calculation of the activation function. Embedded MAC units, also readily available in current generations of FPGAs, are used to implement these operations. The wiring and interconnect between MAC units determines the data flow and it usually mimic that of the neural network structure. Multiplexers and registers are used to facilitate the data transfer between processing elements. In addition, an important factor to consider is the bitwidth of the accumulation process that will be dedicated for the accumulation result which is used as input to the activation function. This depends on the targeted accuracy and can be again resolved with high-level simulations. There are a number of possible options in implementing the activation function. The first choice is through high-level synthesis resulting in the logic structure that directly implements the activation function. However, for functions other than the step-function this is complex and would consume a lot of hardware resources and in a circuit with high latency. A more hardware-efficient way is to approximate the function using piecewise linear approximation, or a look-up table (LUT) to store the activation functions results for the input range. A drawback though for the use of LUT's is that it utilizes memory resources and this reduces the available space that is left for the weights and input data. The LUT method involves deciding the precision of the stored values in the same reasoning as the ANN weights, which is the bit-width and arithmetic representation. Obviously, certain properties of the activation function such as symmetry, saturation and oddity could be exploited. Resource sharing can be utilized as well, by sharing the LUT memory between neurons that complete their MAC operation in different times.

- *Opportunities for parallelism*

ANN's provide a lot of opportunities for parallelism because of their parallel processing capabilities, especially in feed-forwards networks. Each neuron in one layer performs independently from the others, so all neurons in that layer can process data in parallel. Furthermore a neuron may process more than one input at the same time. This is determined by the number of available hardware resources and how they can be distributed for the neurons in the network. ANNs can have a variety of topologies which affects the amount of parallelism that can be achieved and the data flow. A fully connected network operates slower

than a partially connected network because there is more dependency between layers in fully connected networks. As the ANN connectivity increases, the network becomes increasingly demanding in terms of the required communication between neurons. Communication becomes the bottleneck in the implementation, as the wiring and interconnectivity increase the design complexity. This is further elevated on FPGA designs, as their routing and wiring are fixed. Hence, hardware implementations of ANNs could significantly benefit from the Network on Chip paradigm [187].

Christos Kyrkou

Christos Kyrkou

REFERENCES

- [1] Branislav Kisacanin, Shuvra S. Bhattacharyya, and Sek Chai, *Embedded Computer Vision*. London, UK: Springer, 2009.
- [2] Embedded Vision Alliance. [Online]. <http://www.embedded-vision.com/>
- [3] Cha Zhang and Zhengyou Zhang, "A Survey of Recent Advances in Face Detection," Microsoft Research, Redmond USA, Technical Report June 2010.
- [4] Erik Hjelmas and Boon Kee Low, "A Survey of Recent Advances in Face detection," *Journal of Computer Vision and Image Understanding*, vol. 83, no. 3, 2001.
- [5] S. Munder and D.M. Gavrila, "An Experimental Study on Pedestrian Classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1863-1868, 2006.
- [6] Changjian Gao and Shih-Lien Lu, "Novel FPGA based Haar classifier face detection algorithm acceleration," in *International Conference on Field Programmable Logic and Applications*, 2008, pp. 373-378.
- [7] Chun He, Alexandros Papakonstantinou, and Deming Chen, "A novel SoC architecture on FPGA for ultra fast face detection," in *Proceedings of the 2009 IEEE international conference on Computer design*, 2009, pp. 412-418.
- [8] Lijing Zhang and Yingli Liang, "A fast method of face detection in video images," in *International Conference on Advanced Computer Control*, 2010, pp. 490-494.
- [9] S. Meister, B. Jahne, and D. Kondermann, "An outdoor stereo camera system for the generation of Real-World benchmark datasets," *Optical Engineering*, 2011.
- [10] Sparsh Mittal, "A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems," *Journal of Computer Aided Engineering and Technology*, 2014.
- [11] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12)*, 2012, pp. 47-56.
- [12] Arian Maghazeh, Unmesh Bordoloi, Petru Eles, and Zebo Peng, "General Purpose Computing on Low-Power Embedded GPUs : Has It Come of Age?," in *Int. Conf. on Embedded Computer Systems (SAMOS XIII)*, Samos, 15-18 July 2013, pp. 1-10.
- [13] W.J. MacLean, "An Evaluation of the Suitability of FPGAs for Embedded Vision Systems," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2005, p. 131.
- [14] B. Heisele, A. Verri, and T. Poggio, "Learning and Vision Machines," *Proceedings of the IEEE, Visual Perception: Technology and Tools*, vol. 90, no. 7, pp. 1164-1177, 2002.
- [15] Paul Viola and Michael Jones, "Robust real-time face detection," *International Journal of Computer Vision*, pp. 137-154, May 2004.
- [16] Corinna Cortes and Vladimir Vapnik, "Support-Vector Networks," *Journal of Machine Learning*, vol. 20, no. 3, pp. 273-297, September 1995.
- [17] Sergios Theodoridis and Konstantinos Koutroumbas, *Pattern Recognition*, 3rd ed.:

Academic Press, 2008.

- [18] Open CV Library Intel Corporation, Santa Clara, CA. [Online]. <http://opencv.org/>
- [19] Christopher J. C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining and Knowledge Discovery*, vol. 2, pp. 121-167, 1998.
- [20] Constantine Papageorgiou and Tomaso Poggio, "A Trainable System for Object Detection," *International Journal on Computer Vision*, vol. 38, no. 1, pp. 15-33, June 2000.
- [21] Anil K. Jain, Robert P.W. Duin, and Jianchang Mao, "Statistical Pattern Recognition: A Review," *IEEE Transactions on Pattern Analysis and Machine Learning*, vol. 22, no. 1, pp. 4-37, January 2000.
- [22] Ming-Hsuan Yang, David J. Kriegman, and Narendra Ahuja, "Detecting Faces in Images: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 1, pp. 34-58, January 2002.
- [23] Laura Sanchez Lopez, "Local Binary Patterns applied to Face Detection and Recognition," Universitat Politecnica de Catalunya, Barcelona, Technical Report November 2010.
- [24] Anders Kjaer-Nielsen, "Real-time Vision using FPGAs, GPUs and Multi-core CPUs," University of Southern Denmark, Doctoral Dissertation 2010.
- [25] R. Ronen et al., "Coming challenges in microarchitecture and architecture," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 325-340, March 2001.
- [26] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA '12)*, 2012, pp. 47-56.
- [27] B. Sharma, R. Thota, N. Vydyanathan, and Amit Kale, "Towards a Robust, Real-time Face Processing System using CUDA-enabled GPUs," in *Int. Conf. on High Performance Computing (HiPC)*, 2009, pp. 368-377.
- [28] J. D. Owens, M. Houston, D. Luebke, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, 2008.
- [29] J. D. Owens et al., "A Survey of General-Purpose Computation on Graphics Hardware," Eurographics state of the art report 2005.
- [30] Cristian Grozea, Zorana Bankovic, and Pavel Laskov, "FPGA vs. Multi-Core CPUs vs. GPUs: Hands-on Experience with a Sorting Application," in *Facing the Multi-Core Challenge: Conf. for Young Scientists at the Heidelberger Akademie der Wissenschaften*, 2010.
- [31] Ron Sass and Andrew G. Schmidt, *Embedded Systems Design with Platform FPGAs*, 1st ed. USA: Morgan Kaufmann - Elsevier, 2010.
- [32] Jai Gopal Pandley, Abhijit Karmakar, and Chandra Shektar, "Platform-Based Design Approach for Embedded Vision Applications," *Journal of Image and Graphics*, vol. 1, no. 1, pp. 1-6, March 2013.
- [33] R. Dobai and L. Sekanina, "Towards evolvable systems based on the Xilinx Zynq platform," in *IEEE International Conference on Evolvable Systems*, 2013, pp. 89-95.

- [34] Yoav Freund and Robert E. Schapire, "A decision-theoretic generalization of on-line learning," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119-139, August 1997.
- [35] C.P. Papageorgiou, M. Oren, and T. Poggio, "A general framework for object detection," in *International Conference on Computer Vision*, 1998, pp. 555-562.
- [36] Yoav Freund and Robert E. Schapire, "A short introduction to boosting," *Japanese Society for Artificial Intelligence*, vol. 14, no. 5, pp. 771-780, 1999.
- [37] Lutz Hamel, *Knowledge Discovery with Support Vector Machines*. New York, USA: Wiley-Interscience, 2009.
- [38] Bernhard Scholkopf and Alexander J. Smola, *Learninig with kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Cambridge, MA, USA: MIT Press, 2001.
- [39] Vladimir N. Vapnik, *The Nature of Statistical Learning Theory*. New York, USA: Springer-Verlag, 1995.
- [40] Hyeran Bryun and Seong-Whan Lee, "Applications of Support Vector Machines for Pattern Recognition: A Survey," in *International Workshop on Pattern Recognition with Support Vector Machines*, 2002, pp. 213-236.
- [41] B. Scholkopf, C.J. Burges, and A.J. Smola, *Advances in kernel methods*. Cambridge: MIT Press, 1999.
- [42] C.M. Bishop, *Pattern Recognition and Machine Learning*. Secaucus, NJ, USA: Springer-Verlag, 2006.
- [43] Sabri Boughorbel, Jean-Philippe Tarel, and Nozha Boujemaa, "Conditionally Positive Definite Kernels for SVM Based Image Recognition," in *IEEE International Conference on Multimedia and Expo*, 2005, pp. 113-116.
- [44] Chih-Chung Chang and Chih-Jen Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 1-27, 2011.
- [45] John C. Platt, "Fast training of support vector machines using sequential minimal optimization," in *Advances in kernel methods.*: MIT Press, 1999.
- [46] MATLAB Release 2010b, The MathWorks, Inc., Natick, Massachusetts, United States.
- [47] Sang-Hun Yoon, Chun-Gi Lyuh, Ik-Jae Chun, Jung-Hee Suk, and Tae Moon Roh, "A New Support Vector Compression Method Bsed on Singular Value Decomposition," *ETRI Journal*, vol. 33, no. 4, pp. 652-655, August 2011.
- [48] Zhang Jing, Zhang Xue-dong, and Ha Seok-wun, "A Novel Approach Using PCA and SVM for Face Detection," in *International Conference on Natural Computation*, 2008, pp. 29-33.
- [49] Kiri L. Wagstaff, Michael Kocurek, Dominic Mazzoni, and Benyang Tang, "Progressive refinement for support vector machines," *Data Mining and Knowledge Discovery*, vol. 20, no. 1, pp. 53-69, January 2010.
- [50] Tom Downs, Kevin E Gates, and Annette Masters, "Exact Simplification of Support Vector Solutions," *Journal of Machine Learning Research*, vol. 2, 2001.
- [51] Christopher J.C. Burges, "Simplified support vector decision rules," in *International*

Conference on Machine Learning, 1996, pp. 71-77.

- [52] Christopher J.C. Burges and Bernhard Schölkopf, "Improving the Accuracy and Speed of Support Vector Machines," in *Advances in Neural Information Processing Systems*, Denver, CO, USA, 1997, pp. 375-381.
- [53] Benyang Tang and Dominic Mazzoni, "Multiclass Reduced-Set Support Vector Machines," in *Int. Conf. on Machine Learning*, Pittsburgh, Pennsylvania, 2006, pp. 921-928.
- [54] Bernd Heisele, Thomas Serre, Sam Prentice, and Tomaso Poggio, "Hierarchical classification and feature reduction for fast face detection with support vector machines," *Pattern Recognition*, pp. 2007-2017, 2003.
- [55] I. Kukenys and B. McCane, "Classifier cascades for support vector machines," in *International Conference on Image and Vision Computing*, 2008, pp. 1-6.
- [56] Yong Ma and Xiaoqing Ding, "Face Detection Based on Cost-Sensitive Support Vector Machines," in *First International Workshop on Pattern Recognition with Support Vector Machines*, 2002, pp. 260-267.
- [57] S. Romdhani, P. Torr, B. Scholkopf, and A. Blake, "Computationally Efficient Face Detection," in *International Conference on Computer Vision*, 2001, pp. 695-700.
- [58] Sami Romdhani, Philip Torr, Bernhard Schölkopf, and Andrew Blake, "Efficient face detection by a cascaded support-vector machine expansion," *Royal Society of London Proceedings Series A*, vol. 460, no. 2051, pp. 3283-3297, November 2004.
- [59] Hichem Sahbi, Donald Geman, and Nozha Boujemaa, "Face detection using coarse-to-fine support vector classifiers," in *International Conference on Image Processing*, 2001, pp. 925-928.
- [60] I. Buciu, C. Kotropoulos, and I. Pitas, "Combining support vector machines for accurate face detection," in *International Conference on Image Processing*, 2001, pp. 1054-1057.
- [61] Navneet Dalal and Bill Triggs, "Histograms of oriented gradients for human detection," in *IEEE Conf. on Computer Vision and Pattern Recognition*, 2005, pp. 886-893.
- [62] Timo Ojala, Matti Pietikainen, and David Harwood, "A comparative study of texture measures with classification based on featured distributions," *Pattern Recognition*, vol. 26, no. 1, pp. 51-59, 1996.
- [63] A. Hadid, M. Pietikainen, and T. Ahonen, "A Discriminative Feature Space for Detecting and Recognizing Faces," in *IEEE Conf. on Computer Vision and Pattern Recognition*, 2004, pp. 797-804.
- [64] B. Froba and A. Ernst, "Face detection with the modified census transform," in *IEEE Int. Conf. on Automatic Face and Gesture Recognition*, 2004, pp. 91-96.
- [65] T. Ahonen, A. Hadid, and M. Pietikainen, "Face description with local binary patterns An Application to face recognition," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 28, no. 12, pp. 2037-2041, 2006.
- [66] Caifeng Shan, Shaogang Gong, and Peter W. McOwan, "Facial expression recognition based on Local Binary Patterns: A comprehensive study," *Image and Vision Computing*, vol. 27, no. 6, pp. 803-816, 2009.
- [67] Mu Yadong, Yan Shuicheng, Yi Liu, T. Huang, and Zhou Bingfeng, "Discriminative

- local binary patterns for human detection in personal album," in *IEEE Conf. on Computer Vision and Pattern Recognition*, 2008, pp. 1-8.
- [68] Li Liu, Lingjun Zhao, Yunli Long, Gangyao Kuang, and Paul Fieguth, "Extended local binary patterns for texture classification," *Image and Vision Computing*, vol. 30, no. 2, pp. 86-99, February 2012.
- [69] Jiri Trefny and Jiri Matas, "Extended Set of Local Binary Patterns for Rapid Object Detection," in *Computer Vision Winter Workshop*, 2010.
- [70] Matti Pietikainen, Hadid Abdenour, Guoying Zhao, and Timo Ahonen, *Computer Vision Using Local Binary Patterns.*: Springer, 2011.
- [71] David A. Forsyth and Jean Ponce, *Computer Vision: A Modern Approach.*: Prentice Hall Professional Technical Reference, 2002.
- [72] M. S. Sadri et al., "An FPGA Based Fast Face Detector," in *Global Signal Processing Expo & Conference*, Santa Clara, CA, USA, 2008.
- [73] Allam Mousa, "Canny Edge-Detection Based Vehicle Plate Recognition," *International Journal of Signal Processing, Image Processing and Pattern Recognition*, vol. 5, no. 3, pp. 1-7, September 2012.
- [74] M. Raman and H. Aggarwal, "Study and Comparison of Various Image Edge Detection Techniques," *International Journal of Image Processing*, vol. 3, no. 1, pp. 1-12, 2009.
- [75] Oswaldo Ludwig Jr, David Delgado, Valter Goncalves, and Urbano Nunes, "Trainable Classifier-Fusion Schemes An Application to Pedestrian Detection," in *International IEEE Conference On Intelligent Transportation Systems*, 2009, pp. 432-437.
- [76] E. Osuna, R. Freund, and F. Firosi, "Training support vector machines: an application to face detection," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997, pp. 130-136.
- [77] Emanuele Trucco and Alessandro Verri, *Introductory Techniques for 3-D Computer Vision*. USA: Prentice Hall, 1998.
- [78] Chen Yen-Kuang, E. Li, and Tong Xiaofeng, "Parallelization of AdaBoost algorithm on multi-core processors," in *IEEE Workshop on Signal Processing Systems*, 2008, pp. 275-280.
- [79] A. Ranjan and M. Malik, "Parallelizing a Face Detection and Tracking System for Multi-Core Processors," in *Conf. on Computer and Robot Vision*, 2012, pp. 290-297.
- [80] D. Oro, C. Fernandez, J.R. Saeta, X. Martorell, and J. Hernando, "Real-time GPU-based Face Detection in HD Video Sequence," in *IEEE int. Conf. on Computer Vision Workshops*, 2011, pp. 530-537.
- [81] Y. Hanai, Y. Hori, J. Nishimura, and T. Kuroda, "A versatile recognition processor employing Haar-like features and cascade classifiers," in *IEE Int. Solid-State Circuits Conf.*, 2009, pp. 148-149.
- [82] T. Theocharides, N. Vijaykrishnan, and M. J. Irwin, "A parallel architecture for hardware face detection," in *Annual Symposium on Emerging VLSI Technologies and Architectures*, 2006.
- [83] M. Hiromoto, H. Sugano, and R. Miyamoto, "Partially Parallel Architecture for Adaboost-Based Detection with Haar-Like Features," *IEEE Transactions on Circuits*

and *Systems for Video Technology*, vol. 19, no. 1, pp. 41-52, January 2009.

- [84] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fpga-based face detection system using haar classifiers," in *ACM/SIGDA international symposium on Field Programmable Gate Arrays*, New York, USA, 2009, pp. 103-112.
- [85] Y. Wei, X. Bing, and C. Chareonsak, "Fpga implementation of adaboost algorithm for detection of face biometrics," in *EEE International Workshop on Biomedical Circuits and Systems*, 2004.
- [86] Y. Shi, F. Zhao, and Z. Zhang, "Hardware implementation of adaboost algorithm and verification," in *Advanced Information Networking and Applications*, 2008, pp. 343-346.
- [87] Hung-Chih Lai, Marios Savvides, and Chen Tsuhan, "Proposed FPGA Hardware Architecture for High Frame Rate ($\gg 100$ fps) Face Detection Using Feature Cascade Classifiers," in *IEEE International Conference on Biometrics: Theory, Applications, and Systems*, 2007.
- [88] Xilinx, San Jose, CA, "Xilinx University Program," [Online]. <http://www.xilinx.com/univ/>
- [89] Compaq Corp., Palo Alto, CA, "The CACTI Toolset," [Online]. <http://research.compaq.com/wrl/people/jouppi/CACTI.html>
- [90] Fabien Moutarde, Bogdan Stanculescu, and Amaury Breheret, "Real-time visual detection of vehicles and pedestrians with new efficient adaBoost features," in *2nd Workshop on Planning, Perception and Navigation for Intelligent Vehicles (PPNIV)*, at *2008 IEEE International Conference on Intelligent Robots Systems (IROS 2008)*, 2008.
- [91] Hichem Sahbi, Donald Geman, and Pietro Perona, "A Hierarchy of Support Vector Machines for Pattern Detection," *Journal of Artificial Intelligence Research*, vol. 7, pp. 2087-2123, 2006. [Online]. *Journal of Artificial Intelligence Research*
- [92] I. Biasi, A. Boni, and A. Zorat, "A reconfigurable parallel architecture for SVM classification," in *IEEE International Joint Conference on Neural Networks*, 2005, pp. 2867-2872.
- [93] Jian-xiong Dong, Adam Krzyzak, and Ching Y. Suen, "Fast SVM Training Algorithm with Decomposition on Very Large Data Sets," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, no. 4, pp. 603-618, April 2005.
- [94] A. Boni and F. Bardi, "Intelligent Hardware for Identification and Control of Non-Linear Systems with SVM," in *European Symposium on Artificial Neural Networks*, 2001, pp. 75-80.
- [95] C. Papageorgiou and T. Poggio, "Trainable Pedestrian Detection system," *International Journal of Computer Vision*, pp. 15-33, 2000.
- [96] M. Oren, C. Papageorgiou, P. Sinha, E. Osuna, and T. Poggio, "Pedestrian detection using wavelet templates," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997, pp. 193-199.
- [97] D. Roth and S. Agrawal, "Learning a Sparse Representation for Object Detection," in *Int. Conf on Computer Vision*, 2002, pp. 113-130.
- [98] Bernd Heisele, Tomaso Poggio, and Massimiliano Pontil, "Face Detection in Still Gray Images," Center for Biological and Computational Learning, MIT, Cambridge, A.I.

Memo 2000.

- [99] Bernd Heisele, "Visual Object Recognition with Supervised Learning," *IEEE Intelligent Systems*, vol. 18, no. 3, pp. 38-42, June 2003.
- [100] S. Dey, M. Kedia, N. Agrawal, and A. Basu, "Embedded Support Vector Machine: Architectural Enhancements and Evaluation," in *International Conference on VLSI Design*, 2007, pp. 685-690.
- [101] R. Pedersen and M. Schoeberl, "An Embedded Support Vector Machine," in *International Workshop on Intelligent Solutions in Embedded Systems*, 2006, pp. 1-11.
- [102] A. Boni, F. Pianegiani, and D. Petri, "Low-Power and Low-Cost Implementation of SVMs for Smart Sensors," *IEEE Transactions on Instrumentation and Measurement*, vol. 56, no. 1, pp. 39-44, February 2007.
- [103] Hai-xiang Zhao and Frederic Magoules, "Parallel Support Vector Machines on Multi-core and Multiprocessor Systems," in *Int. Conf. on Artificial Intelligence and Applications*, 2010.
- [104] Sergio Herrero-Lopez, John R. Williams, and Abel Sanchez, "Parallel multiclass classification using SVMs on GPUs," in *3rd Workshop on General-Purpose Computation on GPUs*, 2010, pp. 2-11.
- [105] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer, "Fast support vector machine training and classification on graphics processors," in *International conference on Machine learning*, 2009, pp. 104-111.
- [106] Austin Carpenter. (2009) CUSVM: A CUDA Implementation of Support Vector Machines. Report. [Online]. <http://patternsonscreen.net/cuSVM.html>
- [107] Sebastian Bauer, Sebastian Kohler, Konrad Doll, and Ulrich Brunsmann, "FPGA-GPU Architecture for Kernel SVM Pedestrian Detection," in *Computer Vision and Pattern Recognition Workshops*, 2010, pp. 61-68.
- [108] H. P. Graf et al., "A Massively Parallel Digital Learning Processor," in *Annual Conference on Neural Information Processing Systems (NIPS)*, 2008, pp. 529-536.
- [109] Markos Papadonikolakis and Christos Savvas Bouganis, "Novel Cascade FPGA Accelerator for Support Vector Machines Classification," *Transactions on Neural Networks and Learning Systems*, vol. 23, no. 7, pp. 1040-1052, 2012.
- [110] D. Anguita, S. Ridella, and S. Rovetta, "Circuitual implementation of support vector machines," *Electronics Letters*, vol. 34, pp. 1596-1597, 1998.
- [111] M. Papadonikolakis and C. Bouganis, "A Heterogeneous FPGA Architecture for Support Vector Machine Training," in *Field-Programmable Custom Computing Machines (FCCM)*, 2010, pp. 211-214.
- [112] R. Genov and G. Gauwengerghs, "Kerneltron: Support VectorMachine'in Silicon," *IEEE Transactions on Neural Networks*, vol. 14, pp. 1426-1434, 2003.
- [113] D. Anguita, A. Boni, and S. Ridella, "A Digital Architecture for Support Vector Machines: Theory, Algorithm, and FPGA Implementation," *IEEE Transactions on Neural Networks*, vol. 14, no. 5, pp. 993-1009, September 2003.
- [114] D. Esteve, and D. Martinez R. Reyna, "An integrated vision system: object detection and localization," in *Workshop on Design of Mixed-Mode Integrated Circuits and*

Applications, 1999, pp. 118-121.

- [115] O. Pina-Ramirez, R. Valdes-Cristerna, and O. Yanez-Suarez, "An FPGA implementation of linear kernel support vector machines," in *IEEE Int. Conf. on Reconfigurable Computing and FPGA's*, 2006, pp. 1-6.
- [116] T. Kryjak, M. Komorkiewicz, and M. Gorgon, "FPGA implementation of real-time head-shoulder detection using local binary patterns, SVM and foreground object detection," in *Conf. on Design and Architectures for Signal and Image Processing*, 2012, pp. 1-8.
- [117] Davood Mahmoodi, Ali Soleimani, Hossein Khosravi, and Mehdi Taghizadeh, "FPGA Simulation of Linear and Nolinear Support Vector Machine," *Journal of Software Engineering and Applications*, pp. 320-328, 2011.
- [118] Roberto Reyna, Dominique Houzet, and Marie-France Albenge, "Implementation of the SVM Neural Network Generalization Function for Image Processing," in *IEEE International Workshop on Computer Architectures for Machine Perception*, 2000, p. 147.
- [119] Roberto Reyna-Rojas, Dominique Houzet, Daniela Dragomirescu, Florent Carlier, and Salim Ouadjaout, "Object Recognition System-on-Chip Using the Support Vector Machines," *EURASIP Journal on Advances in Signal Processing*, pp. 993-1004, 2005.
- [120] T. Groleat, M. Arzel, and S. Vaton, "Hardware Acceleration of SVM-based traffic classification on FPGA," in *Int. Wireless Communications and Mobile Computing Conf.*, 2012, pp. 443-449.
- [121] Marta Ruiz-Llata and Mar Yebenes-Calvino, "FPGA Implementation of Support Vector Machines for 3D Object Identification," in *International Conference on Artificial Neural Networks*, 2009, pp. 467-474.
- [122] M. Ruiz-Llata, G. Guarnizo, and M. Yébenes-Calvino, "FPGA implementation of a support vector machine for classification and regression," in *International Conference on Neural Networks*, 2010, pp. 1-5.
- [123] S. Cadambi et al., "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines," in *IEEE International Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2009, pp. 115-122.
- [124] D. Anguita, A. Boni, and S. Ridella, "Learning for nonlinear support vector machines suited for digital VLSI," *Electronics Letters*, vol. 35, pp. 1349-1350, 1999.
- [125] A. Boni and D. Anguita, "Digital least squares support vector machines," *Neural Processing Letters*, vol. 18, pp. 65-72, 2003.
- [126] F.M. Khan, M.G. Arnold, and W.M. Pottenger, "Finite Precision Analysis of Support Vector Machine Classification in Logarithmic Number Systems," in *Euromicro Symposium on Digital System Design*, 2004, pp. 254-261.
- [127] F.M. Khan, M.G. Arnold, and W.M. Pottenger, "Hardware-based support vector machine classification in logarithmic number systems," in *IEEE International Symposium on circuits and systems*, 2005, p. 5154.
- [128] A. Boni and A. Zorat, "FPGA Implementation of Support Vector Machines with Pseudo-Logarithmic Number Representation," in *International Joint Conference on Neural Networks*, 2006, pp. 618-624.

- [129] D. Anguita, A. Ghio, S. Pischiutta, and S. Ridella, "A Hardware-friendly Support Vector Machine for Embedded Automotive Applications," in *International Joint Conference on Neural Networks*, 2007, pp. 1360-1364.
- [130] D. Anguita, A. Ghio, and S. Pischiutta, "A learning machine for resource-limited adaptive hardware," in *Second NASA/ESA Conference on Adaptive Hardware and Systems*, 2007, pp. 571-576.
- [131] Alessandro Ghio and Stefano Pischiutta, "A Support Vector Machine based pedestrian recognition system on resource-limited hardware architectures," in *Research in Microelectronics and Electronics Conference*, 2007, pp. 161-163.
- [132] J. Gomes Filho, M. Raffo, M. Strum, and Wang Jiang Chau, "A general-purpose dynamically reconfigurable SVM," in *Programmable Logic Conference*, 2010, pp. 107-112.
- [133] D. Anguita, S. Pischiutta, S. Ridella, and D. Sterpi, "Feed-forward support vector machine without multipliers," *IEEE Transactions on Neural Networks*, vol. 17, p. 1328, 2006.
- [134] K. Irick, M. DeBole, V. Narayanan, and A. Gayasen, "A Hardware Efficient Support Vector Machine Architecture for FPGA," in *International Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 304-305.
- [135] M. Papadonikolakis and C. Bouganis, "A novel FPGA-based SVM classifier," in *International Conference on Field-Programmable Technology*, 2010, pp. 283-286.
- [136] B. Heisele, T. Poggio, and M. Pontil, "Face detection in still gray images," *Technical Report*, 2000.
- [137] Bernd Heisele, Purdy Ho, Jane Wu, and Tomaso Poggio, "Face recognition: component-based versus global approaches," *Computer Vision and Image Understanding*, vol. 91, no. 1/2, pp. 6-21, July 2003.
- [138] SMO Algorithm MATLAB. Mathworks, MATLAB online documentation. [Online]. <http://www.mathworks.com/help/toolbox/bioinfo/ref/svmtrain.html>
- [139] CBCL Face Database #1, MIT Center for Biological and Computation Learning. [Online]. <http://cbcl.mit.edu/software-datasets/FaceData2.html>
- [140] UIUC Image Database for Car Detection. [Online]. <http://l2r.cs.uiuc.edu/~cogcomp/Data/Car/>
- [141] CBCL PEDESTRIAN DATABASE #1", MIT Center for Biological and Computation Learning. [Online]. <http://iris.usc.edu/Vision-Users/OldUsers/bowu/DatasetWebpage/dataset.html>
- [142] CMU and MIT Face Database. [Online]. http://vasc.ri.cmu.edu/idb/html/face/frontal_images/
- [143] USC Pedestrian Detection Test Set C. [Online]. <http://iris.usc.edu/Vision-Users/OldUsers/bowu/DatasetWebpage/dataset.html>
- [144] Microblaze Soft Processor. Xilinx, San Jose, CA. [Online]. <http://www.xilinx.com/tools/microblaze.htm>
- [145] H. Sahbi and Nozha Boujemaa, "Coarse-to-fine support vector classifiers for face detection," in *International Conference on Pattern Recognition*, 2002, pp. 359-362.

- [146] JAFFE - The Japanese Female Facial Expression Database. [Online].
<http://www.kasrl.org/jaffe.html>
- [147] Bao Face Database. [Online]. <http://www.facedetection.com/facedetection/datasets.htm>
- [148] M. Murat Dundar and Jumbo Bi, "Joint optimization of cascaded classifiers for computer aided detection," in *Conf. on Computer Vision and Pattern Recognition*, 2007, pp. 1-8.
- [149] Minmin Chen, Zhixiang Xu, Kilian Weinberger, Olivier Chapelle, and Dor Kedem, "Classifier Cascade for Minimizing Feature Evaluation Cost," in *Int. Conf. on Artificial Intelligence and Statistics*, 2012.
- [150] Shireen Elhabian, Khaled El-Sayed, and Sumaya Ahmed, "Moving object detection in spatial domain using background removal techniques - state-of-art," *Recent Patents on Computer Science*, vol. 1, no. 1, pp. 32-34, 2008.
- [151] J. Kovac, P. Peer, and F Solina, "Human skin color clustering for face detection," in *EUROCON International Conference on Computer as Tool*, 2003, pp. 144-148.
- [152] Lin Hwei-Jen, Yen Shwu-Huey, Yeh Jih-Pin, and Lin Meng-Ju, "Face Detection Based on Skin Color Segmentation and SVM Classification," in *Second International Conference on Secure System Integration and Reliability Improvement*, 2008.
- [153] Che Ming and Chang Yisong, "A Hardware/Software Co-design of a Face Detection Algorithm Based on FPGA," in *International Conference on Measuring Technology and Mechatronics Automation*, 2010, pp. 109-112.
- [154] T. Darrell, G. Gordon, M. Harville, and J. Woodfill, "Integrated Person Tracking Using Stereo, Color, and Pattern Detection," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1998, p. 601.
- [155] J.G. Wang, E.T. Lim, and R. Venkateswarlu, "Stereo head/face detection and tracking," in *International Conference on Image Processing*, 2004, pp. 605-608.
- [156] Sergey Kosov, Kristina Scherbaum, Kamil Faber, Thorsten Thormählen, and Hans-Peter Seidel, "Rapid stereo-vision enhanced face detection," in *IEEE international conference on Image processing*, 2009, pp. 1217-1220.
- [157] H. Wu, K. Suzuki, T. Wada, and Q. Chen, "Accelerating Face Detection by Using Depth Information," in *3rd Pacific Rim Symposium on Advances in Image and Video Technology, Lecture Notes in Computer Science*, pp. 657-667.
- [158] S. Anila and N. Devarajan, "Simple and Fast Face Detection System Based on Edges," *International Journal of Universal Computer Sciences*, vol. 1, no. 2, pp. 54-58, March 2010.
- [159] G. Hetzel, B. Leibe, P. Levi, and B. Schiele, "3D object recognition from range images using local feature histograms," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2001, pp. 394-399.
- [160] B. Browatzki, J. Fischer, B. Graf, H.H. Bulthoff, and C. Wallraven, "Going into depth: Evaluating 2D and 3D cues for object classification on a new, large-scale object dataset," in *IEEE International Conference on Computer Vision Workshops*, 2011, pp. 1189-1195.
- [161] P. Moreels and P. Perona, "Evaluation of feature detectors and descriptors based on 3D

- objects," in *10th Int. Conf. on Computer Vision*, 2005, pp. 800-807.
- [162] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz, "Fast point feature histograms (FPFH) for 3D registration," in *IEEE international conference on Robotics and Automation*, 2009, pp. 1848-1853.
- [163] Bastian Steder, Radu Bogdan Rusu, Kurt Konolige, and Wolfram Burgard, "NARF: 3D Range Image Features for Object Recognition," in *In Workshop on Defining and Solving Realistic Perception Problems in Personal Robotics*, 2010.
- [164] J. Liebelt, C. Schmid, and Klaus Schertler, "Viewpoint-independent object class detection using 3D Feature Maps," in *IEEE Conference on Pattern Recognition*, 2008, pp. 23-28.
- [165] D. Roobaert and M.M. Van Hulle, "View-based 3D object recognition with support vector machines," in *IEEE Signal Processing Society Workshop*, 1999, pp. 77-84.
- [166] Dongil Han, Jongho Choi, Jae-Il Cho, and Dongsu Kwak, "Design and VLSI implementation of high-performance face-detection engine for mobile applications," in *IEEE International Conference on Consumer Electronics*, 2011, pp. 705-706.
- [167] N. Farrugia, F. Mamalet, S. Roux, Fan Yang, and M. Paindavoine, "Fast and Robust Face Detection on a Parallel Optimized Architecture Implemented on FPGA," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 4, pp. 597-602, April 2009.
- [168] Rob McCready, "Real-Time Face Detection on a Configurable Hardware System," in *International Workshop on Field-Programmable Logic and Applications*, 2000, pp. 157-162.
- [169] Christophe Garcia and Manolis Delakis, "Convolutional face finder: a neural architecture for fast and robust face detection," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2007, pp. 1408-1423.
- [170] Stavros Hadjitheophanous, Christos Ttofis, Athinodoros Georghiades, and Theocharis Theocharides, "Towards hardware stereoscopic 3D reconstruction a real-time FPGA computation of the disparity map," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2010, pp. 1743-1748.
- [171] Christos Ttofis, Stavros Hadjitheophanous, Athinodoros Georghiades, and Theocharis Theocharides, "Edge-Directed Hardware Architecture for Real-Time Disparity Map Computation," *IEEE Transactions on Computers*, 2012.
- [172] Camera Calibration Toolbox for Matlab. [Online].
http://www.vision.caltech.edu/bouguetj/calib_doc/
- [173] Image Sequence Analysis Test Site - Set 1 (Night vision car stereo sequences).
Multimedia Imaging Technology portal - University of Auckland. [Online].
<http://www.mi.auckland.ac.nz/>
- [174] D.D. Le and S. Satoh, "Feature selection by AdaBoost for SVM-based face detection," *Information Technology Letters*, 2004.
- [175] Alper Yilmaz, Omar Javed, and Mubarak Shah, "Object tracking: A survey," *ACM Computing Surveys*, vol. 38, no. 6, December 2006.
- [176] Woo Joon Han and Il Song Han, "Neuromorphic Visual Information Processing - Bio-Inspired Vision," *International Journal of Computer Theory and Engineering*, vol. 5, no.

- 1, pp. 52-55, February 2013.
- [177] T. Serre and T. Poggio, "A neuromorphic approach to computer vision," *Communications of the ACM*, vol. 53, no. 10, pp. 54-61, October 2010.
- [178] QingXiang Wu, T.M. McGinnity, Liam Maguire, Rngtai Cai, and Meigui Chen, "A visual attention model based on hierarchical spiking neural networks," *Neurocomputing*, 2012.
- [179] Pierre Greisen, Manuel Lang, Simon Heinzle, and Aljosa Smolic, "Algorithm and VLSI Architecture for Real-Time 1080p60 Video Retargetting," in *Eurographics conference on High-Performance Graphics*, 2012, pp. 57-66.
- [180] D. Kerr, D. Coleman, T.M. McGinnity, and M. Clogenson, "Biologically inspired intensity and range image feature extraction," *International Conference on Neural Networks*, pp. 1-8, August 2013.
- [181] G. Chatziparaskevas, A. Brokalakis, and I. Papaefstathiou, "An FPGA-based parallel processor for Black-Scholes option pricing using finite differences schemes," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 709-714.
- [182] Kuikang Cao and Haibin Shen, "Scalable SM Processor and its Application to Non-linear Channel Equalization," in *Pacific-Asia Conference on Circuits, Communications, and System*, 2009, pp. 206-209.
- [183] R.K. Sevakula and N.K. Verma, "Wavelet transforms for fault detection using SVM in Power Systems," in *IEEE Int. Conf. on Power Electronics, Drives and Energy Systems*, 2012, pp. 1-6.
- [184] Frank Y. Shih, *Image Processing and Pattern Recognition: Fundamentals and Techniques.*: Wiley-IEEE Press, May 2010.
- [185] E. Kakouli, V. Soteriou, and T. Theocharides, "Intelligent Hotspot Prediction for Network-on-Chip-Based Multicore Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 418-431, March 2012.
- [186] T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin, and W. Wolf, "Embedded Hardware Face Detection," in *International Conference on VLSI Design*, 2004.
- [187] T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin, and V. Srikantam, "A Generic Reconfigurable Neural Network Architecture implemented as a Network on Chip," in *IEEE International System on Chip Conference*, 2004, pp. 191-194.