Technical University of Crete
Electronic and Computer Engineering Department
Microprocessor & Hardware Laboratory

Diploma Thesis

Implementation of the Receptive Field Cooccurrence Histograms Algorithm for Object Detection on FPGA

Author:
Savvas Papaioannou

Committee:

Supervisor:
Assistant Professor:
Yannis Papaefstathiou

Professors:
Apostolos Dollas
Michael Zervakis

June 30, 2011

Abstract

Object Detection and Recognition is a central and important challenge in the field of computer vision and autonomous robotics. In robotic systems there is often a need for detecting and locating certain objects in natural environments in real time. Although a significant amount of Object Detection and Recognition algorithms have been developed and reported in the bibliography, demonstrating outstanding and state of the art results, they fail to be applied real time in robotics. Due to their high computational complexity, they require a significant amount of processing power, they are time consuming and not power-efficient, things that make them impractical for real-time robotic systems.

In this thesis we studied the Receptive Field Cooccurrence Histograms Algorithm (RFCH) for object detection [1] with the aim to improve its performance. After analyzing the algorithm's hot-spots we describe in this thesis how we can increase its performance using the hardware accelerators that we designed. Moreover, we continued with software/hardware co-design and we prototyped an embedded system on FPGA for the specific algorithm.

If we knew what we were doing, it wouldn't be called research, would it?

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter we make a brief introduction into Computer Vision and its applications and later we introduce the Field Programmable Gate Arrays (FPGAs) and discuss how their use becomes beneficial to computer vision applications.

1.1 Computer vision science

Computer vision (or machine vision) is the science and technology that enables machines to solve particular tasks by extracting information from an image. Computer Vision is a diverse and relatively new, although rapidly developing, field of study with strong scientific and industrial support. As a technological discipline, computer vision seeks to apply its theories and models to the construction of computer vision systems. Examples of applications of computer vision include systems for:

- Autonomous robots and driver-less vehicles.
- Visual surveillance, people counting, video content analysis and visual sensor networks.
- Face recognition, image classification and Object detection/recognition.
- Medical image analysis and topographical modeling.
- Artificial visual perception.

Computer Vision is also related to a lot of other fields such as:

- Artificial intelligence
- Image processing and image analysis
- Pattern recognition
- Neural Networks
- Mathematics

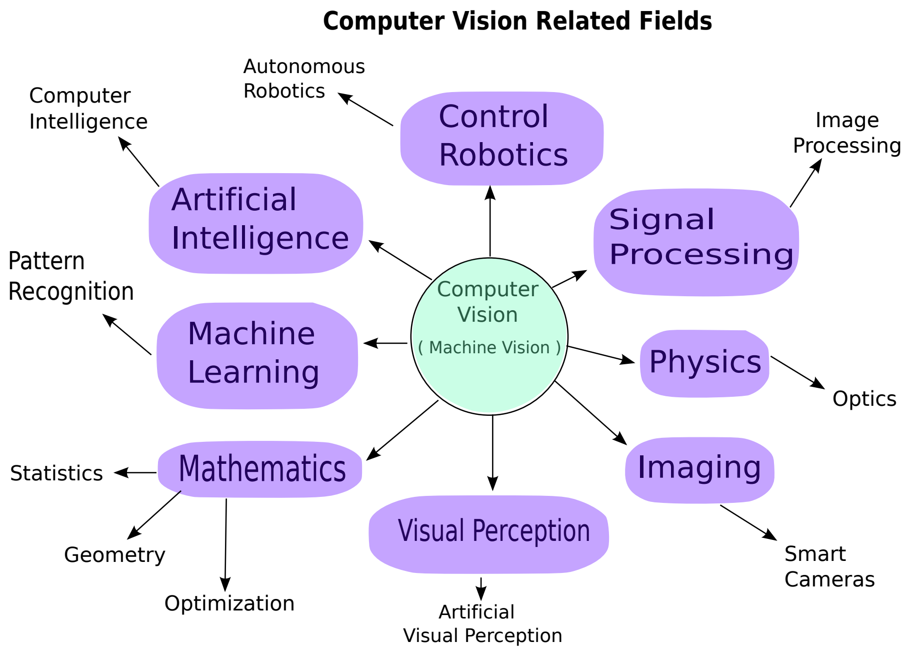


Figure 1.1: Relation between robotic vision and various other fields

1.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. As opposed to Application Specific Integrated Circuits (ASICs) where the device is custom built for the particular design, FPGAs can be programmed to fit the desired application or functionality requirements - hence ”field-programmable”. The FPGA configuration is generally specified using a hardware description language (HDL). FPGAs can be used to implement any logical function that an ASIC could perform. The ability to update the functionality after shipping, the partial re-configuration of the portion of the design and the low non-recurring engineering costs relative to an ASIC design offer advantages for many applications. The figure below illustrates Altera’s Stratix II (left) and Xilinx’s Virtex-5 (right) FPGA logic blocks [2].

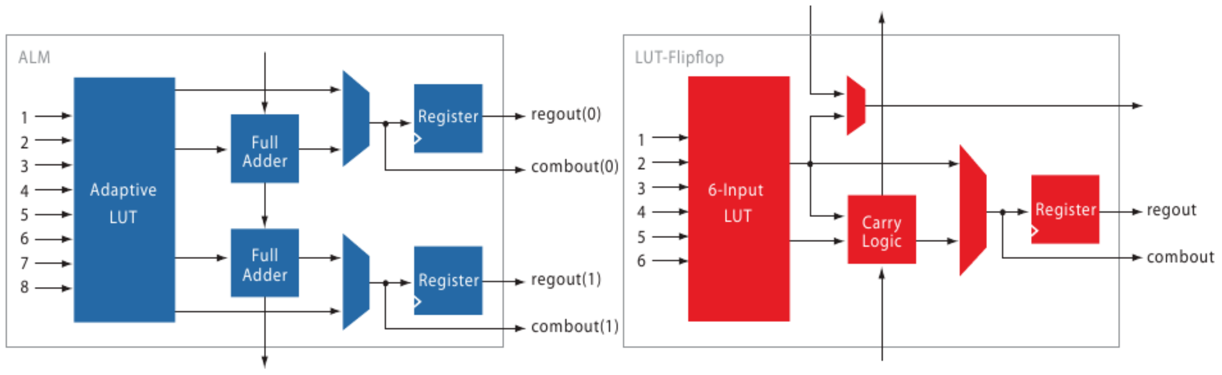
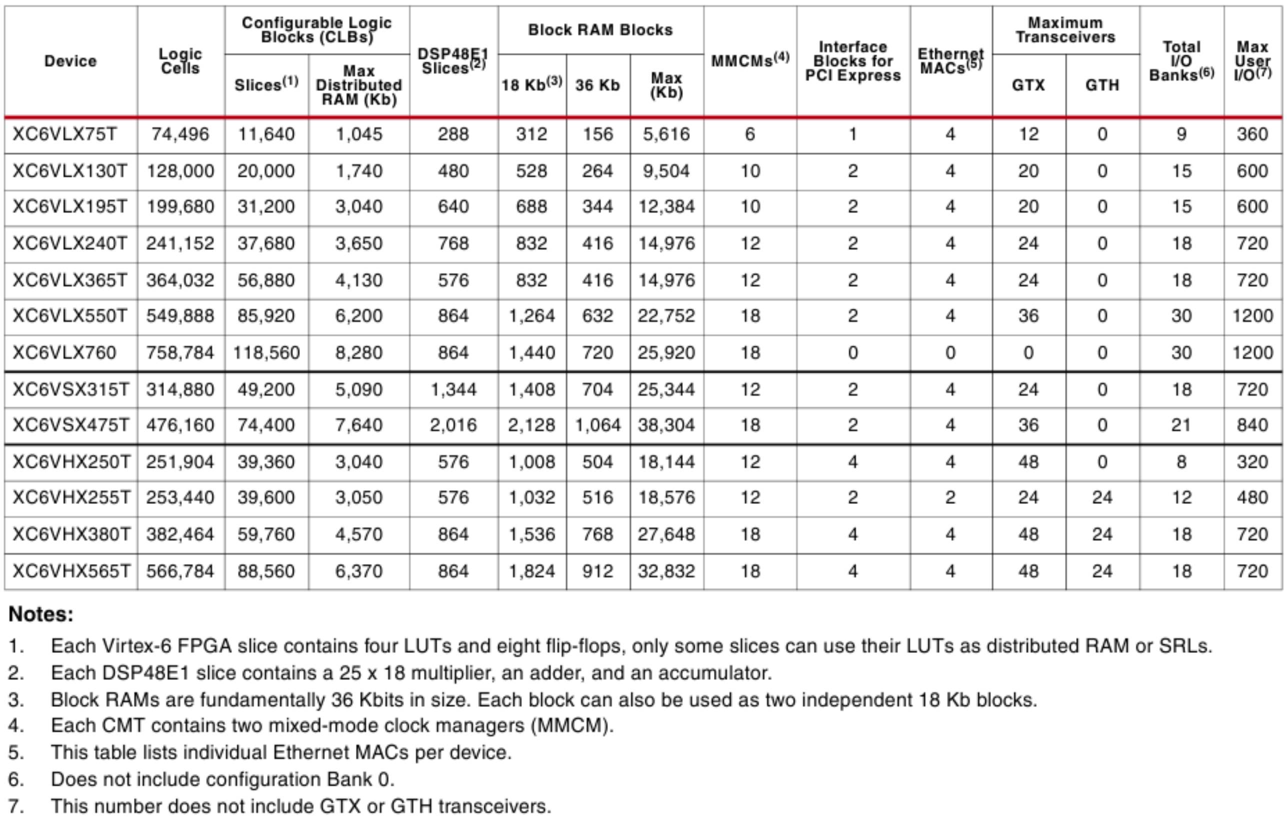


Figure 1.2: FPGA configurable logic blocks

The Xilinx Virtex-5 logic element (also called a LUT flipflop pair), consists of a basic 6-LUT, carry logic, and a single register as shown in figure above.

Modern FPGAs combine the CLB's with configurable block RAM blocks, DSP blocks and with one or more microprocessors embedded within the FPGA's logic fabric, which makes the FPGAs a complete "system on a programmable chip" suitable for a variety of applications. The table below summarizes the Xilinx Virtex-6 family features [3].



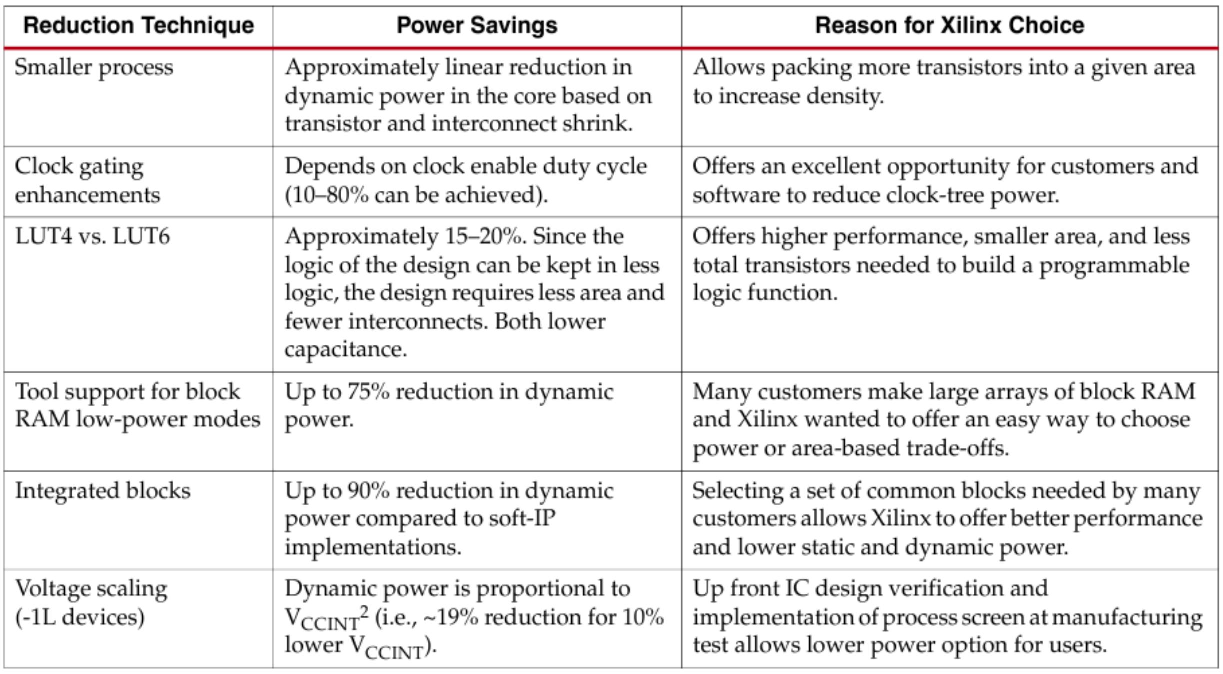
Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices(2)	Block RAM Blocks			MMCMs(4)	Interface Blocks for PCI Express	Ethernet MACs(5)	Maximum Transceivers		Total I/O Banks(6)	Max User I/O(7)
		Slices(1)	Max Distributed RAM (Kb)		18 Kb(3)	36 Kb	Max (Kb)				GTX	GTH		
XC6VLX75T	74,496	11,640	1,045	288	312	156	5,616	6	1	4	12	0	9	360
XC6VLX130T	128,000	20,000	1,740	480	528	264	9,504	10	2	4	20	0	15	600
XC6VLX195T	199,680	31,200	3,040	640	688	344	12,384	10	2	4	20	0	15	600
XC6VLX240T	241,152	37,680	3,650	768	832	416	14,976	12	2	4	24	0	18	720
XC6VLX365T	364,032	56,880	4,130	576	832	416	14,976	12	2	4	24	0	18	720
XC6VLX550T	549,888	85,920	6,200	864	1,264	632	22,752	18	2	4	36	0	30	1200
XC6VLX760	758,784	118,560	8,280	864	1,440	720	25,920	18	0	0	0	0	30	1200
XC6VSX315T	314,880	49,200	5,090	1,344	1,408	704	25,344	12	2	4	24	0	18	720
XC6VSX475T	476,160	74,400	7,640	2,016	2,128	1,064	38,304	18	2	4	36	0	21	840
XC6VHX250T	251,904	39,360	3,040	576	1,008	504	18,144	12	4	4	48	0	8	320
XC6VHX255T	253,440	39,600	3,050	576	1,032	516	18,576	12	2	2	24	24	12	480
XC6VHX380T	382,464	59,760	4,570	864	1,536	768	27,648	18	4	4	48	24	18	720
XC6VHX565T	566,784	88,560	6,370	864	1,824	912	32,832	18	4	4	48	24	18	720

Notes:

1. Each Virtex-6 FPGA slice contains four LUTs and eight flip-flops, only some slices can use their LUTs as distributed RAM or SRLs.
2. Each DSP48E1 slice contains a 25 x 18 multiplier, an adder, and an accumulator.
3. Block RAMs are fundamentally 36 Kbits in size. Each block can also be used as two independent 18 Kb blocks.
4. Each CMT contains two mixed-mode clock managers (MMCM).
5. This table lists individual Ethernet MACs per device.
6. Does not include configuration Bank 0.
7. This number does not include GTX or GTH transceivers.

Table 1.1: Virtex-6 FPGA Feature Summary

A key feature of FPGAs is the ability to reduce the dynamic power consumption according to the developer's needs. A couple of dynamic power reduction techniques are the Clock gating which is offered in all modern FPGAs and the ability to operate certain IPs (such as block RAMs) in low-power mode in the design . Some dynamic power reduction techniques which are offered in Xilinx's Virtex-6 and Spartan-6 FPGAs [4] are summarized in the figure below.



Reduction Technique	**Power Savings**	**Reason for Xilinx Choice**
Smaller process	Approximately linear reduction in dynamic power in the core based on transistor and interconnect shrink.	Allows packing more transistors into a given area to increase density.
Clock gating enhancements	Depends on clock enable duty cycle (10–80% can be achieved).	Offers an excellent opportunity for customers and software to reduce clock-tree power.
LUT4 vs. LUT6	Approximately 15–20%. Since the logic of the design can be kept in less logic, the design requires less area and fewer interconnects. Both lower capacitance.	Offers higher performance, smaller area, and less total transistors needed to build a programmable logic function.
Tool support for block RAM low-power modes	Up to 75% reduction in dynamic power.	Many customers make large arrays of block RAM and Xilinx wanted to offer an easy way to choose power or area-based trade-offs.
Integrated blocks	Up to 90% reduction in dynamic power compared to soft-IP implementations.	Selecting a set of common blocks needed by many customers allows Xilinx to offer better performance and lower static and dynamic power.
Voltage scaling (-1L devices)	Dynamic power is proportional to V_{CCINT}^2 (i.e., ~19% reduction for 10% lower V_{CCINT}).	Up front IC design verification and implementation of process screen at manufacturing test allows lower power option for users.

Table 1.2: Dynamic Power Reduction Techniques in Xilinx Virtex-6 and Spartan-6 FPGAs

1.3 Computer Vision on Hardware

As we have discussed in the previous paragraph, computer vision is a very important science not only because of its useful applications but also because of the plethora of other sciences that depends on it and vice versa.

In the recent years Computer vision is embracing a new research focus (commonly known as Robotic Vision) in which the aim is to develop visual skills for robots that would allow them to interact with a dynamic, realistic environment.

Nowadays a lot of computer vision algorithms exist that can carry out the last, demonstrating state of the art results.

Some of them use global methods meaning that they calculate the object representation on all available image data. Global methods include detection via histograms,co-occurrence histograms, weak classifiers with boosting and others.

In contrast, local feature-based methods only capture the most representative parts of an object. In this category the detection is based on the extraction of powerful discriminative features of the image that describe the desired object.

Despite the different way in which the above algorithms approach the detection problem, it seems that all of them give promising results. However, the vast majority of them, due to their high computational complexity, can not be applied in Real-Time computer vision systems such as autonomous (mobile) robotic systems since they need to process large amount of data in real-time and since some of them need to operate on low-power.

To overcome these problems new kinds of vision algorithms need to be developed which would run in real-time and subserve the robot's goals. However, until that happens many computer vision algorithms are implemented in hardware as application-specific integrated circuits (ASICs), while others are implemented on reconfigurable logic hardware such as Field-programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs).

By implementing a computer vision algorithm in hardware we can gain a significant speedup over the software version, because we can exploit the by nature parallelism of the computer vision task. For example suppose that we want to calculate the histogram of a grey-scale image. In software running on a general purpose CPU, we would have to scan sequentially the whole image, in order to produce the histogram and thus several CPU cycles would be required, for the completion of this task. On the other hand, a possible hardware implementation would be to split the image in four chunks and pass each chunk into custom processors that work in parallel. Finally, when the processing of the four chunks would be completed the final image histogram would be formed from each processor's result. That way a significant speedup would be achieved. Moreover, by implementing a specific task in hardware we can achieve extra performance by using techniques like pipelining and double buffering, which can not be used in the software version of the task.

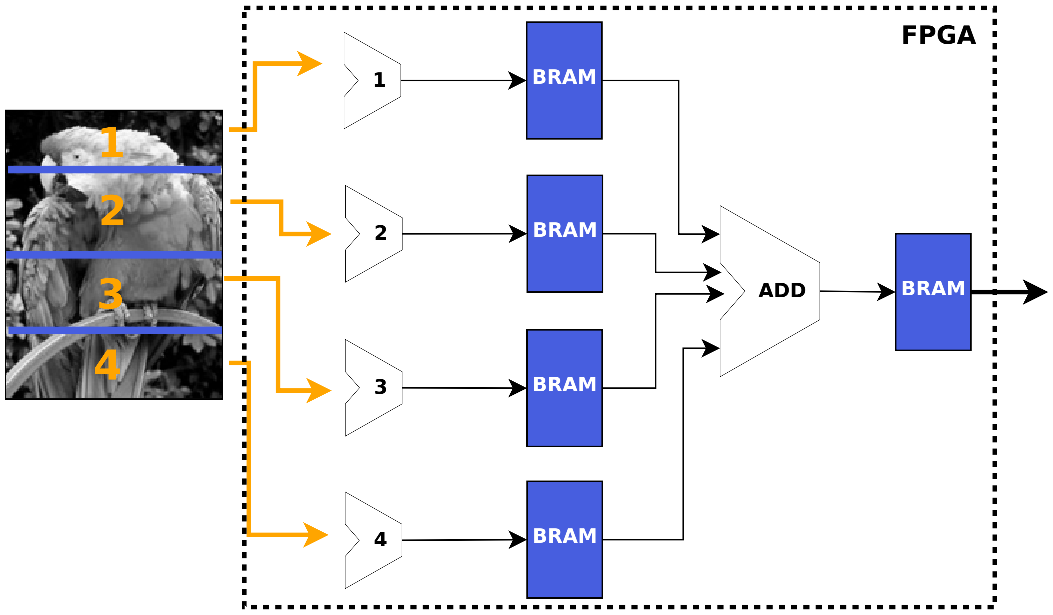


Figure 1.3: Hardware Acceleration of computer vision tasks

The other important benefit of the hardware implementation, is the designer's ability to control the power consumption of the chip, by using dynamic power reduction techniques. It is often necessary for some tasks to be run in low-power mode, on mobile platforms such as mobile robotic systems that operate on batteries. A possible solution to this problem would be the implementation of the later tasks in hardware. Both ASICs and FPGAs can achieve power consumption reduction.

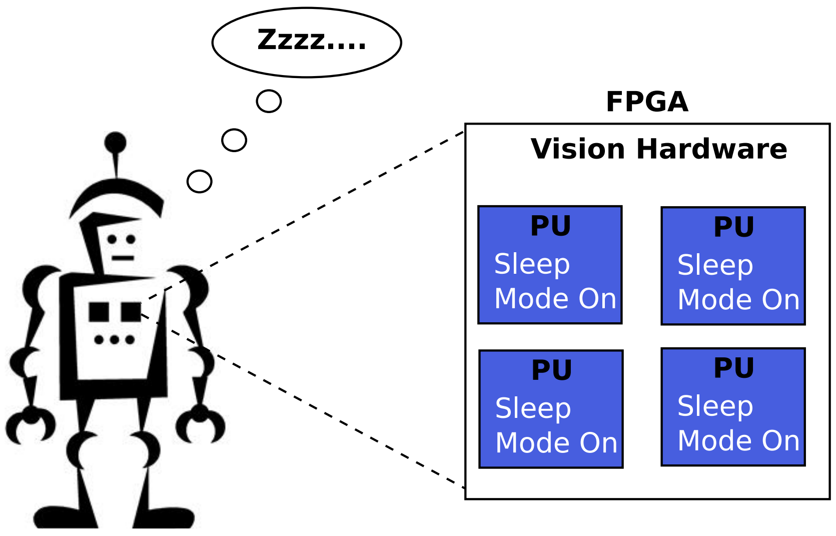


Figure 1.4: Power consumption reduction on hardware

Chapter 2

Object Detection

In this chapter we introduce fundamental concepts about object detection and recognition and later we continue by presenting the common object detection and recognition algorithms out there.

2.1 Object Detection vs Object Recognition

Recognizing objects is one of the major research topics in the field of computer vision and autonomous robotics. In robotics, there is often a need for a system that can locate certain objects in the environment - the capability which we denote as “object detection”. Although a significant amount of work has been reported, the proposed methods still differ significantly regarding these two research areas.

An object recognition algorithm is typically designed to classify an object to one of a several predefined classes assuming that the segmentation of the object has already been performed. Commonly, the test images show a single object that is centered in the image and occupies most of the image area. The test image may also have a black background [5], making the task even simpler.

On the other hand, the task for an object detection algorithm is much harder. Its purpose is to search for a specific object in an image of a complex scene. Most of the object recognition algorithms may be used for object detection by using a search window and scanning the image for the object. Regarding the computational complexity, some methods are more suitable for searching than others. Two fundamental goals of object detection algorithms are to identify a known object in a realistic environment and also to determine its location.

2.2 Object Recognition Algorithms

In terms of object recognition, shape-based approaches and appearance-based approaches are commonly used in order to describe the object to be detected.

Shape-based methods represent an object by its shape/contour whereas in appearance-based methods only the appearance is used, which is usually captured by different two-dimensional views of the object-of-interest.

Some of the algorithms below might be able to perform object detection, but are commonly classified as Object Recognition.

2.2.1 Appearance-based algorithms

The authors in [6] proposed a compact representation of the object's appearance, that is for each object of interest, a large set of images is obtained by automatically varying pose and illumination. Then this set is compressed to obtain a low-dimensional subspace, called eigenspace, in which the object is represented as a manifold. Given an unknown input image, the recognition system projects the image to the eigenspace. The object is then recognized based on the manifold it lies on.

In [7] the authors describe an appearance-based system that uses four distinct levels of perceptual grouping to represent 3D objects in a form that allows 3D object recognition. Then in [8] they use an image-trainable, feature-based method based on the previous vision system for object recognition. Their object recognition system combines information from several views of the scene, to obtain object-in-scene hypotheses with higher confidences. Then the hypothesis with the highest confidence is examined, and appropriate action is taken.

The author in [9] proposes an appearance-based Probabilistic approach for solving the object recognition problem achieving state of art results. Specifically the author proposes to model the visual appearance of objects and visual categories via probability density functions. The model is developed on the basis of concepts and results obtained in three different research areas: computer vision, machine learning and statistical physics of spin glasses. It consists of a fully connected Markov random field with energy function derived from results of statistical physics of spin glasses. Markov random fields and spin glass energy functions are combined together via nonlinear kernel functions.

2.2.2 Shape-based algorithms

In [10] the authors extend the method of Fourier descriptors to produce a set of normalized coefficients which are invariant under any affine transformation (translation, rotation, scaling), which they use in order to recognize objects that are described by closed contours. The basic idea is that a closed curve may be represented by a periodic function of a continuous parameter, or alternatively, by a set of Fourier coefficients of this function.

Also the authors in [11] suggest object recognition based on 2D shape recognition. In their framework the shape of the object is represented by its contour (possibly open). The unknown shape is recognized by morphing its contour to known templates stored in a database.

2.3 Object Detection Algorithms

In this category we will describe the algorithms that perform well on the Object detection task. Based on the applied features these algorithms can be sub-divided into two main classes, namely local and global methods.

A local feature is a property of an image (object) located on a single point or on a small region. It is a single piece of information that describes a distinctive property of the object. In local methods various local features are combined in order to obtain a more complex description of the image usually referred to as descriptor, that has high discriminative power.

In contrast, global methods try to cover the information content of the whole image, by using for example histograms of features and other techniques.

2.3.1 Global Methods

Back in 1991, Swain and Ballard [13] demonstrated how RGB color histograms can be used for object recognition. Specifically they showed that color histograms of multicolored objects provide a robust, efficient cue for indexing into a large database of models. Moreover, they demonstrated that color histograms are stable object representations in the presence of occlusion and over change in view, and that they can differentiate among a large number of objects. Also for solving the identification problem, they introduced an important and simple technique called Histogram Intersection, which matches model and image histograms. Additionally, for solving the location problem they have introduced an algorithm called Histogram Backprojection, which performs this task efficiently in crowded scenes.

The authors in [14] generalized this idea to histograms of receptive fields, and computed histograms of either first-order Gaussian derivative operators or the gradient magnitude and the Laplacian operator at three scales. They showed that the appearance of an object is composed of local structure and this local structure can be described and characterized by a vector of local features measured by local operators such as Gaussian derivatives or Gabor filters. Based on joint statistics, the authors developed techniques for the identification of multiple objects at arbitrary positions and orientations in a cluttered scene. Experiments showed that these techniques can identify over 100 objects in the presence of major occlusions.

In 2004 Linde & Lindeberg [15] in their paper presented a set of composed histogram features of higher dimensionality, which gave significantly better recognition performance compared to the histogram descriptors of lower dimensionality that were used in the original papers by Swain & Ballard (1991).

In [16] the author uses multiple low-level attributes such as color, local shape and texture to develop a histogram-based object recognition system.

The authors in [17] use color cooccurrence histograms (CH) for object detection. The color CH keeps track of the number of pairs of certain colored pixels that occur at certain separation distances in image space. The color CH adds geometric information to the normal color histogram, which abstracts away all geometry. They compute model CHs based on images of known objects taken from different points of view. These model CHs are then matched to subregions in test images to find the object. They showed that the algorithm works in spite of confusing background clutter and moderate amounts of occlusion and object flexing.

2.3.2 Local Methods

In [18] Lowe presents the SIFT features, which is a promising approach for detecting objects in natural scenes. However, the method relies on the presence of feature points and, for objects with simple or no texture, this method fails.

Chapter 3

Related Work

In [19] the authors present a novel approach to use FPGA to accelerate the Haar-classifier based face detection algorithm. By using a large number of parallel arithmetic units in the FPGA they achieved real-time performance of face detection having very high detection rate and low false positives.Their implementation is realized on a HiTech Global PCIe card that contains a Xilinx XC5VLX110T FPGA chip. The authors performed a profiling on the software-based face detection application and determined that Haar classifier function costs more than 95% of the total time and thus they populated only the Haar classifier function step onto the FPGA board and left the pre-processing and post-processing on the host PC (personal computer). In the proposed architecture the host microprocessor executes the pre-processing and sends the integral image and image variances to the FPGA accelerator through PCIe bus. The FPGA proceeds with the Haar classifier function, and then sends the detected faces coordinates back to host microprocessor through PCIe. The host microprocessor finishes the face detection algorithm with the post-processing.Their system showed significant speedup over the software application.

The authors in [20] and [21] are dealing with the problem of edge detection, which is a basic step in image processing and one of the first step performed by many computer vision algorithms, in order to identify sharp discontinuities in an image, such as changes in luminosity or in the intensity due to changes in scene structure. The authors in [20] proposed a new self-adapt threshold Canny edge detector and also presented an FPGA implementation of their algorithm suitable for mobile robotic systems. Their hardware implementation uses the Altera Cyclone EP1C60240C8 and can perform the algorithm on a grey-scale image 360x280 in 2.5ms clocked at 27MHz. In [21] the authors present an other implementation of the Canny edge detector that takes advantage of 4-pixel parallel computation, which increases the throughput of the design without increasing the need for on-chip cache memories. They showed increased throughputs for high resolution images and a computation time of 3.09ms for a 1.2Mpixel image on a Spartan-6 FPGA clocked at 200MHz.

In [22] is being described an FPGA implementation based on moment invariants and Kohonen neural networks for object classification that is able to classify objects in real-time. The authors implemented the classification phase in hardware while leaving the training of the Kohonen network into software. According to the paper, the the computation of moment invariants has been implemented in hardware along with a set of sixteen parallel Kohonen neurons for the classification of an unknown object, demonstrating a possible real-time solution for object classification.

The authors in [23] present an FPGA-Based People Detection System. They use JPEG-compressed frames from a network camera to capture the images, and then they send the extracted features to a machine-learning-based detector implemented on a Virtex-II 2V1000 , to carry out the detection process. The system is demonstrated on an automated video surveillance application detecting people accurately at a rate of about 2.5 frames per second using a MicroBlaze processor running at 75 MHz to control the whole sysem behavior and for communicating with dedicated hardware over FSL links.

Also in [24] the authors describe a real-time computer vision system based on a mini-robot equipped with a micro-processor, with a CMOS camera and with an FPGA extension module. The CMOS camera transmits the image into the FPGA where color histograms are calculated by the custom hardware, and then are transferred to the mini-robot's micro-processor for further processing.

Chapter 4

Receptive Field Cooccurrence Histogram for Object Detection

In this chapter we summarize the Receptive Field Cooccurrence Histogram (RFCH) algorithm, which is described in great detail in [1].

4.1 Introduction

A Receptive Field Histogram is a statistical representation of the occurrence of several descriptor responses within an image. Examples of such image descriptors are color intensity, gradient magnitude and Laplace response. If only color descriptors are taken into account, we have a regular color histogram.

A Receptive Field Cooccurrence Histogram (RFCH) is able to capture more of the geometric properties of an object. Instead of just counting the descriptor responses for each pixel, the histogram is built from pairs of descriptor responses. The pixel pairs can be constrained based on, for example, their relative distance. This way, only pixel pairs separated by less than a maximum distance, d_{max} are considered. Thus, the histogram represents not only how common a certain descriptor response is in the image but also how common it is that certain combination of descriptor responses occur close to each other. In other words, a RFCH is a representation of how often pairs of certain filter responses and colors lie close to each other in the image. This means that more geometric information is preserved and the recognition task becomes easier.

The figure below presents the concept of the cooccurrence histogram, of a 3bit (8 colors) greyscale image. In the figure below we search for co-occurrences from left to right with $d_{max} = 1$.

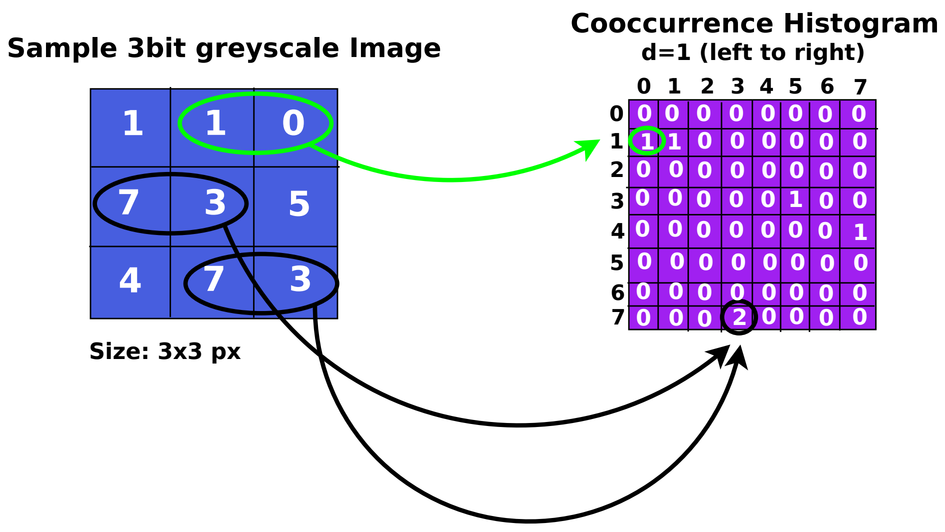


Figure 4.1: Cooccurrence Histogram example

4.2 Algorithm's details

4.2.1 Image Descriptors

The algorithm is able to work with different types of image descriptors such as Color, Gradient magnitude, Laplacian, Gabor, ... and also any mixture of them depending on the task. For object detection a good choice is to use rotationally invariant image descriptors. In particular a mixture of Color, Gradient magnitude and Laplacian descriptors have been proposed for object detection.

4.2.2 Image Quantization

When using histograms as a basis for recognition, computational problems can easily occur if the dimensionality of the histogram is too high. Regular multidimensional receptive field histograms have one dimension for each image descriptor. This makes the histograms huge. For example, a 16-dimensional histogram with 15 quantization levels per dimension contains $15^{16} \approx 10^{19}$ cells in total, while most of the cells will be empty in practice. Building a cooccurrence histogram makes the histogram even bigger, in which case we need about 10^{38} bin entries.

To avoid this problem the algorithm first clusters the input data, so a dimension reduction is achieved. Hence, by choosing the number of clusters he histogram size may be controlled.The cluster centers (N) have a dimensionality equal to the number of image descriptors used. The algorithm is using the K-Means clustering algorithm [25] for the dimension reduction. After the quantization each object ends up with its own cluster scheme in addition to the RFCH calculated on the quantized training image.

When searching for an object in a scene, the image is quantized with the same cluster-centers as the cluster scheme of the object being searched for. Quantization of the search image also has a positive effect on object detection performance. Pixels lying too far from any cluster in the descriptor space are classified as the background and not incorporated in the histogram.

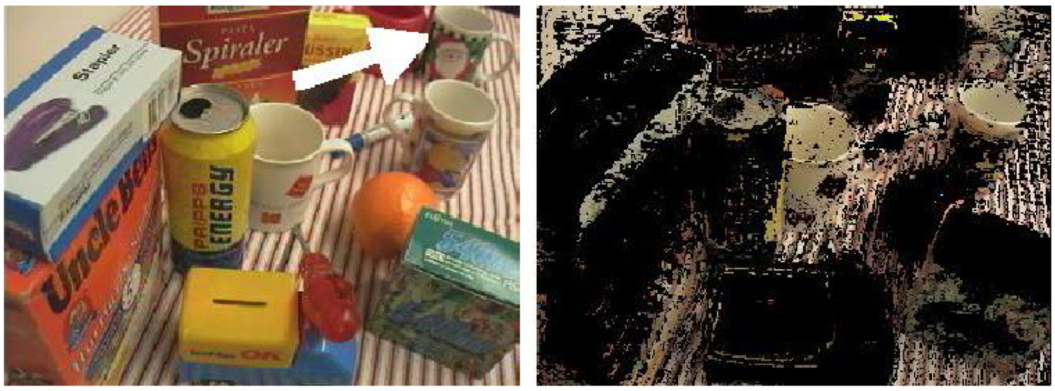


Figure 4.2: Clustering step in RFCH

In the above image (taken from the original RFCH paper [1]) we see what happens when we are searching for the Santa cup. The pixels that lie too far away from their nearest cluster are ignored (set to black in this example).

4.2.3 RFCH building

After the clustering step, the algorithm creates the object's cooccurrence histograms from the clustered descriptor space. In the testing phase the image is scanned using a small search window and the RFCH of the window is calculated. In each scan the RFCH of the window is compared with the object's RFCH.

4.2.4 Histogram Matching

The similarity between two normalized RFCHs is computed as the histogram intersection:

$$\mu(h_1, h_2) = \sum_{n=1}^{N^2} min(h_1[n], h_2[n]) \tag{4.1}$$

where $h_i[n]$ denotes the frequency of receptive field combinations in bin n for image i, quantized into N cluster centers. The higher the value of $\mu(h_1, h_2)$, the better the match between the histograms.
Another popular histogram similarity measure is the x^2 :

$$\mu(h_1, h_2) = \sum_{n=1}^{N^2} \frac{(h_1[n] - h_2[n])^2}{h_1[n] + h_2[n]} \tag{4.2}$$

In this case, the lower value of $\mu(h1, h2)$, the better the match between the histograms. The authors in [1] have found out that x^2 performs much worse than histogram intersection when used for object detection while it performs slightly better on object recognition image databases.

As we mentioned above the RFCH of the object is compared to the window RFCH and the similarity between two RFCHs is computed. The matching vote $\mu(h_{object}, h_{window})$ indicates the likelihood that the window contains the object. Once the entire image has been searched through, a vote matrix provides a hypothesis of the object's location. The most probable location is corresponding to the vote cell with the maximum value.

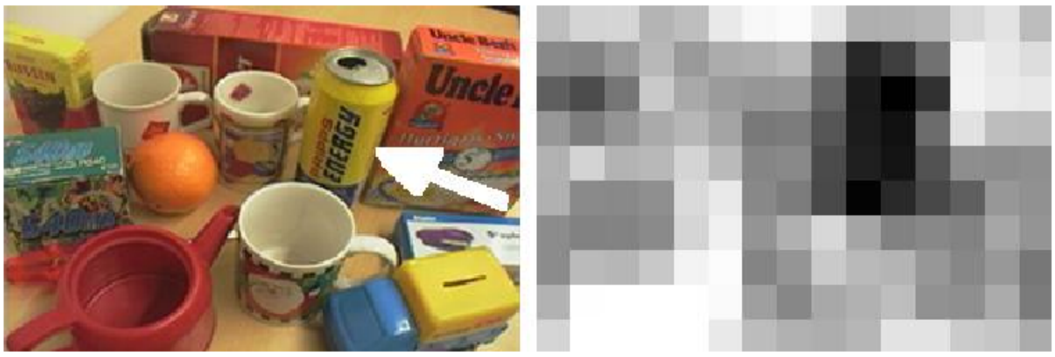


Figure 4.3: RFCH matching

The figure above shows the corresponding vote matrix for the yellow soda can and reveals the object's location.

4.2.5 Free parameters

Number of cluster-centers (N)

The authors state that too few cluster-centers reduce the detection rate, although 80 clusters are sufficient for most descriptor combinations. In our experiments we have used 7 descriptors, and indeed 80 clusters centers are sufficient.

Maximum pixel distance (d_{max})

The parameter d_{max} determines the amount of cooccurrence information. Using $d_{max} = 0$ means no cooccurrence information. Experiments showed that for $d_{max} > 15$ detection rate starts to decrease. We have experimented with $d_{max} = 1..10$ with very good results.

Size of cluster-centers (α)

Pixels that lie outside all of the cluster centers are classified as background and not taken into account. According to the authors, the algorithm performs optimally when $\alpha = 1.5$, however we have evaluated the algorithm with also $\alpha = 2$ without noticing any significant detection rate decrease.

Search window size

This parameter seems to be the drawback of the algorithm. In general this is the main drawback of all the algorithms that are using the "Sliding Window" method in the testing phase, because optimal window size is not known in advance. In addition different image and object sizes might have different optimal search window.

4.3 The RFCH at a glance

The algorithm works in two phases and performs the following steps in order to detect an object :

Training Phase

- Extract Features from the Object
- Calculate Feature Clusters
- Quantize Object
- Create object's RFCH

Detection Phase

- Quantize image with Object's cluster scheme
- Calculate the RFCH for a small window of the image (for all windows)
- Match Object and Image RFCH with histogram intersection (for all windows)
- Take the best match

4.4 Conclusion

RFCH is very a robust algorithm which can detect object despite severe occlusions and cluttered backgrounds. It is also invariant to scale changes, rotation and illumination variations. The algorithm also gives state of the art results with just one training image of each object.

Although the algorithm is designed for robotic applications and is indeed pretty fast, its speed starts to decrease as the number of objects and images increases and thus becomes time consuming and not applicable for real-time applications.

Chapter 5

RFCH Profile Analysis

In this chapter we are going to analyze the code of the algorithm and present its most time consuming parts along with a profile analysis.

5.1 RFCH important functions

The most important functions of the algorithm are listed below:

Features[ ]**=GetImageFeatures(Image**[ ]**)**

This function takes an image and extracts the desired combination of feature descriptors.

ClusterPoint[ ]**=CalculateClusters(Feature**[ ]**)**

This is the function that is responsible for the features clustering. It basically performs an iterative version of the K-Means algorithm. As input it takes the array of features and returns a clustered version of the input array.

BinnedImage[ ]**=ClusterFeatures(Features**[ ]**, CluserPoint**[ ]**)**

This function performs Image quantization. It is quantizing an image according to known (eg. Object's) clusters centers.

RFCH[ ]**=CalculateRFCH(BinnedImage**[ ]**)**

This is the function that creates the receptive field cooccurrence histogram.

Match=MatchRFCHs(RFCH1[ ]**, RFCH2**[ ]**)**

This function performs the histogram intersection, and returns the similarity measurement.

FindObjectInImage(Image[ ]**, Object)**

This function scans the input Image with a sliding window, calculating the RFCH of the window and then compares it with the object's RFCH. When the whole image is scanned it returns the best match along with the coordinates of the window that had the best match. (It calls internally all the above functions exempt CalculateCluster())

The figure below shows the basic steps that the algorithm performs in order to detect one object.

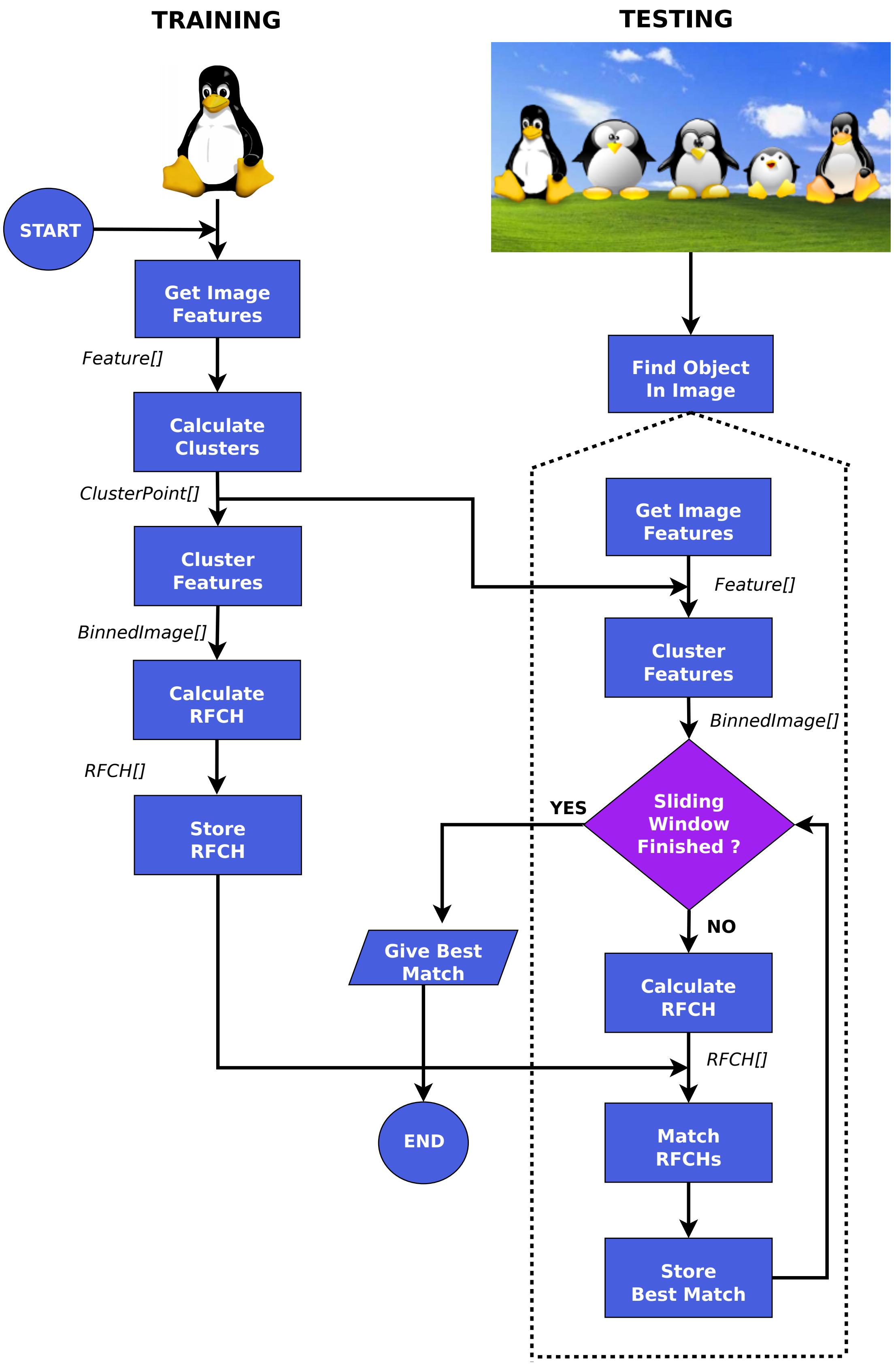


Figure 5.1: RFCH Object Detection steps

5.2 Profiling

In this section we present the profile analysis of the algorithm. In brief, we found out that the most time consuming parts of the algorithm are the below functions:

- CalculateClusters()
- ClasterFeatures()
- CalculateRFCH()

In order to improve the algorithm's performance we decided to implement the above functions in hardware, and leave the rest in software. For the profiling we have used Intel's VTune™ Amplifier XE 2011 [27]. The profiling was performed on an Intel SU7300 ULV CPU @ 1.3GHz in order to capture the scenario in which the algorithm runs on a mobile robotic platform which would usually be equipped with a ULV processor. For the profile analysis that follows we have set the algorithm's free parameters as below:

- Number of features (f) : 7
- Number of Cluster-Centers (N) : 80
- Maximum pixel distance (d_{max}) : 4
- Size of cluster-centers (α) : 1.5
- Search window size : 20
- Image Size : 640 x 480

During this thesis we have contacted all of our experiments using images from CVAP Object Detection Image Database [28], which we have first rescale as 640x480.

Concerning the above three functions we have to mention that:

1. Firstly the Calculate Cluster function is performed only on the training phase of the algorithm and it is applied on training images. Training images contain only one big object inside black background, like figure 5.2. In this algorithm the black color is modeled as null. The Calculate Cluster function works with training images. We will call images like figure 5.2 just objects.

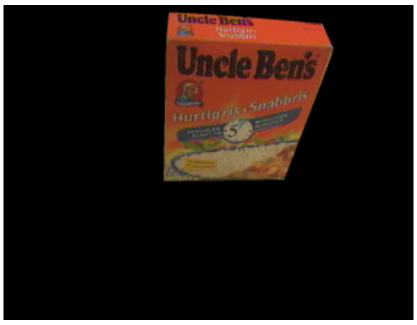


Figure 5.2: Image containing an object used for training

2. The Cluster Feature function is executed in training phase and also in testing phase. In other words if we want to locate one object in one scene this function it will be executed once for the object and once for the scene.

3. The Calculate RFCH function is also executed in the training phase and also in the testing phase. In the training phase this function is executed once for the object, and in the testing this function is executed for many small windows of the scene.

The below tables show the time that algorithm spends in each function. Depending on the test, each function is executed multiple times.

5.2.1 1 Object - 1 Scene

Function Name	**CPU Time (s)**
Calculate Clusters	22.051
Cluster Features	2.061
Calculate RFCH	1.529
Create Image Gauss	0.300
Create Image Gradient Magnitude	0.090
Create Image Laplace	0.060
Normalize	0.051
Match RFCHs	0.050
Create BW Image	0.010

Table 5.1: Profiling: 1 Object - 1 Scene

5.2.2 3 Object - 1 Scene

Function Name	**CPU Time (s)**
Calculate Clusters	38.938
Cluster Features	5.532
Calculate RFCH	2.486
Create Image Gauss	0.599
Match RFCHs	0.380
Create Image Gradient Magnitude	0.190
Create Image Laplace	0.110
Normalize	0.060
Create BW Image	0.041

Table 5.2: Profiling: 3 Object - 1 Scene

5.2.3 10 Object - 1 Scene

Function Name	**CPU Time (s)**
Calculate Clusters	119.198
Cluster Features	18.301
Calculate RFCH	7.438
Create Image Gauss	1.607
Match RFCHs	1.297
Create Image Gradient Magnitude	0.550
Create Image Laplace	0.299
Normalize	0.080
Create BW Image	0.071

Table 5.3: Profiling: 10 Object - 1 Scene

5.2.4 10 Object - 10 Scenes

Function Name	**CPU Time (s)**
Cluster Features	169.043
Calculate Clusters	120.109
Calculate RFCH	56.987
Match RFCHs	12.523
Create Image Gauss	2.098
Create Image Gradient Magnitude	0.989
Create Image Laplace	0.601
Normalize	0.260
Create BW Image	0.195

Table 5.4: Profiling: 10 Object - 10 Scenes

As we can see from the above tables, the functions CalculateClusters(), ClusterFeatures() and CalculateRFCH() are taking up to 97.8% of the algorithms total time. By making the above 3 functions faster, we can significantly improve the performance of the overall algorithm since 97.8% of the time, the algorithm executes the 3 functions mentioned.

5.2.5 Expected Speedup

If we suppose that we accelerate the 3 functions that consume the 97.8% of the algorithm and we leave the other 2.2% as is [1] then we can calculate the expected overall speed up of the algorithm using Amdahl's Law as :

$$Speedup_{overall} = \frac{Execution\ Time_{old}}{Execution\ Time_{new}} = \frac{1}{(1-0.978) + \frac{f_1}{x_1} + \frac{f_2}{x_2} + \frac{f_3}{x_3}} \tag{5.1}$$

where f_1, f_2, f_3 is the fraction of time that CalculateClusters(), ClusterFeatures() and CalculateRFCH() are executed and x_1, x_2, x_3 is the speedup of the re-implemented CalculateClusters(), ClusterFeatures() and CalculateRFCH() respectively.

5.2.6 Software/Hardware Partitioning

Based on the above profile analysis of the algorithm and considering Amdahl's law we decided to partition the algorithm as follows:

Software:

1. Feature Extraction (i.e Create Image Gauss, Create BW Image ... etc)
2. Histogram Intersection (i.e MatchRFCHs)

Hardware:

1. Calculate Clusters
2. Cluster Features
3. Calculate RFCH

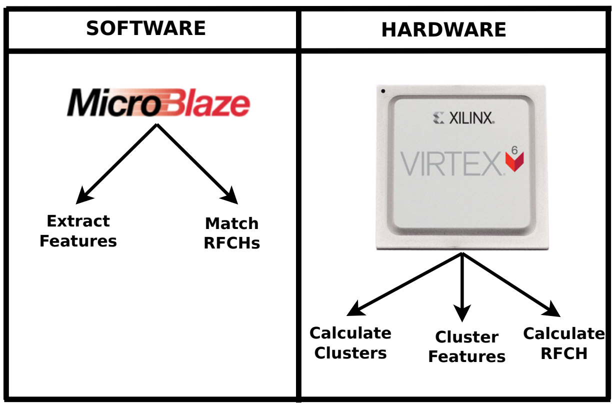


Figure 5.3: Software/Hardware Partitioning

[1]We suppose that functions that take the 2.2% will perform the same as seen on the tables on an equivalent processor or platform.

5.3 RFCH Hot-Spots

In this section we analyze further the algorithms hot-spots.

5.3.1 CalculateClusters

This function is performing an iterative version of the K-Means algorithm, and it has been identified as hot-spot. Lets take a look at the code. Below is the pseudo-code of the iterative K-Means algorithm [26] :

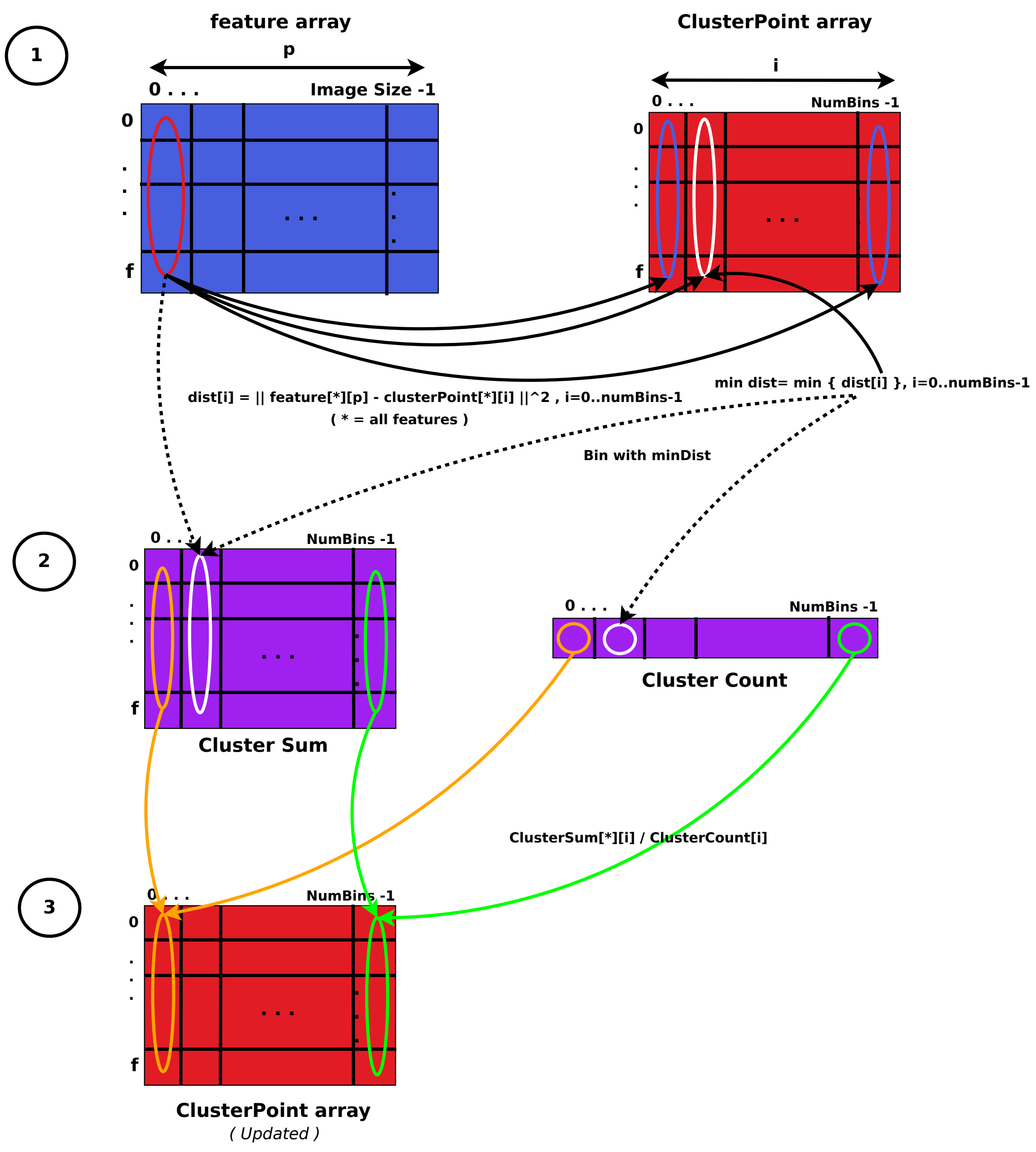
```
{Initialize cluster centers}
repeat
  for $i = 1..n$ do
    {classify the sample $i$ according to nearest cluster center}
  end for
  {recompute cluster centers}
until no change in cluster centers
return cluster centers
```

The computational complexity of the above algorithm is $O(nfNT)$ where n is the number of samples , f is the number of features, N is the number of clusters and T is the number of iterations until convergence.

The figure below shows the algorithm in a graphical way. In our case the samples to be clustered are equal to the size of the image(n=image size), and each one has f features. The algorithm stores this information in the feature array which has size=f x $ImageSize$.

Moreover the number of clusters ($N = numBins$) have dimensionality equal to the number of features (f). The algorithm stores this information to the ClusterPoint array with size=(f x $numBins$).

Additionally the algorithm stores temporary information into two arrays as the below figure shows. In the below figure, steps 1 and 2 consist the classification phase and step 3 is the recalculation of the cluster centers. All the steps are repeated until the algorithm converges.



<u>CalculateClusters Algorithm</u>

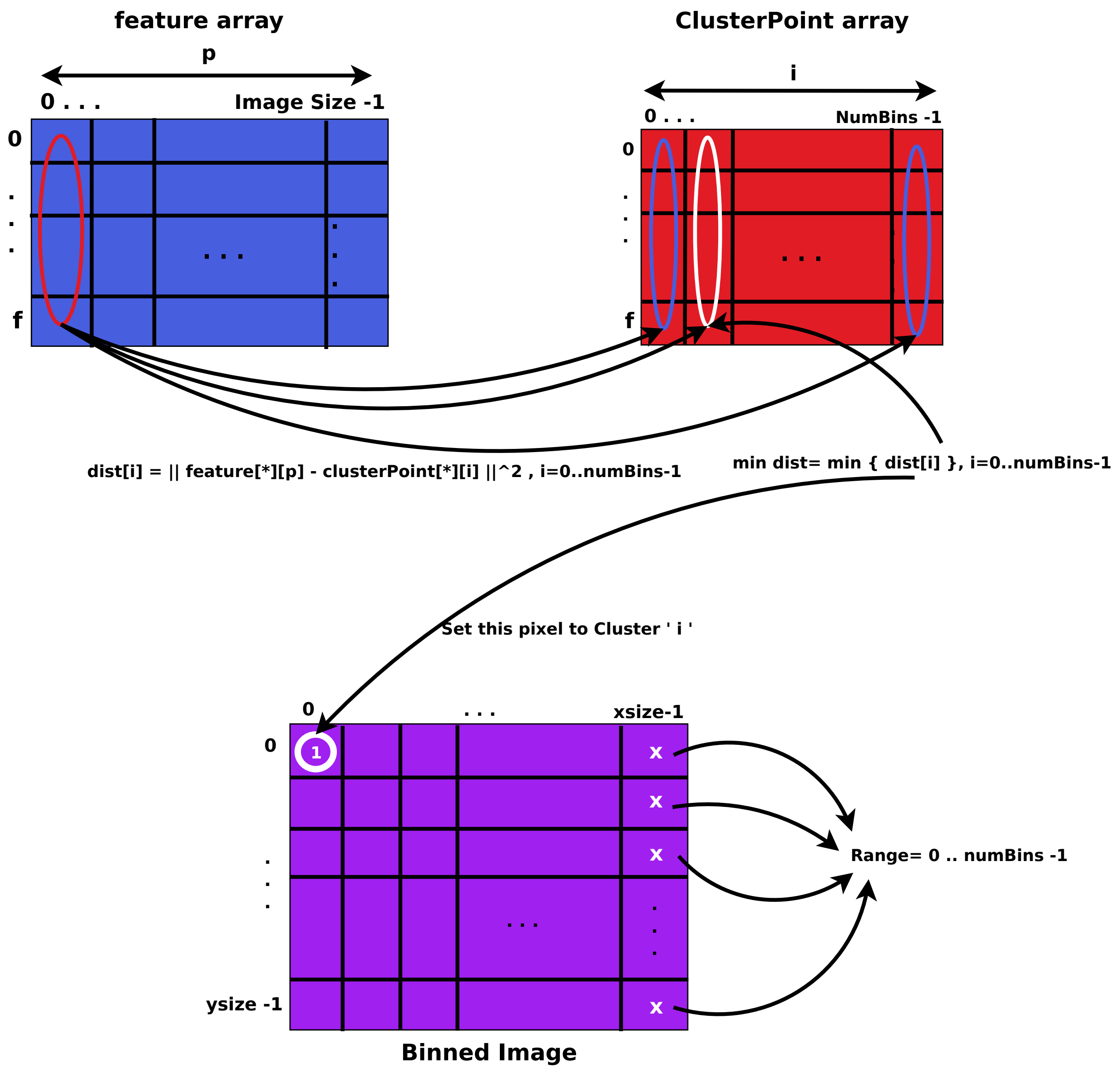
```
for (p=0.. Image Size-1) {
    STEP 1:
    Cluster the feature[*][p] into the nearest cluster , where *=all features

    STEP2:
    for (f=0..numFeature-1) {
    ClusterSum[f][tmp]=feature[f][p], where tmp=bin with the minDist
    }
    ClusterCount[tmp]++ , where tmp=bin with minDIst
}
  STEP 3:
  for (i=0..numBins-1) {
  clusterPoint[*][i]=ClusterSum[*][i] / ClusterCount[i], where *= all features
  }
```

Figure 5.4: Calculate Clusters function visualization

5.3.2 ClusterFeatures

This function is responsible for the quantization of the image according to the precalculated cluster centers. The function has complexity $O(nfN)$ A graphical representation of the function follows in the next figure. Bellow the array binnedImage is the quantized image based on the calculated clusters (ClusterPoint Array). ClusterFeatures is an other hot-spot according to our profiling analysis.



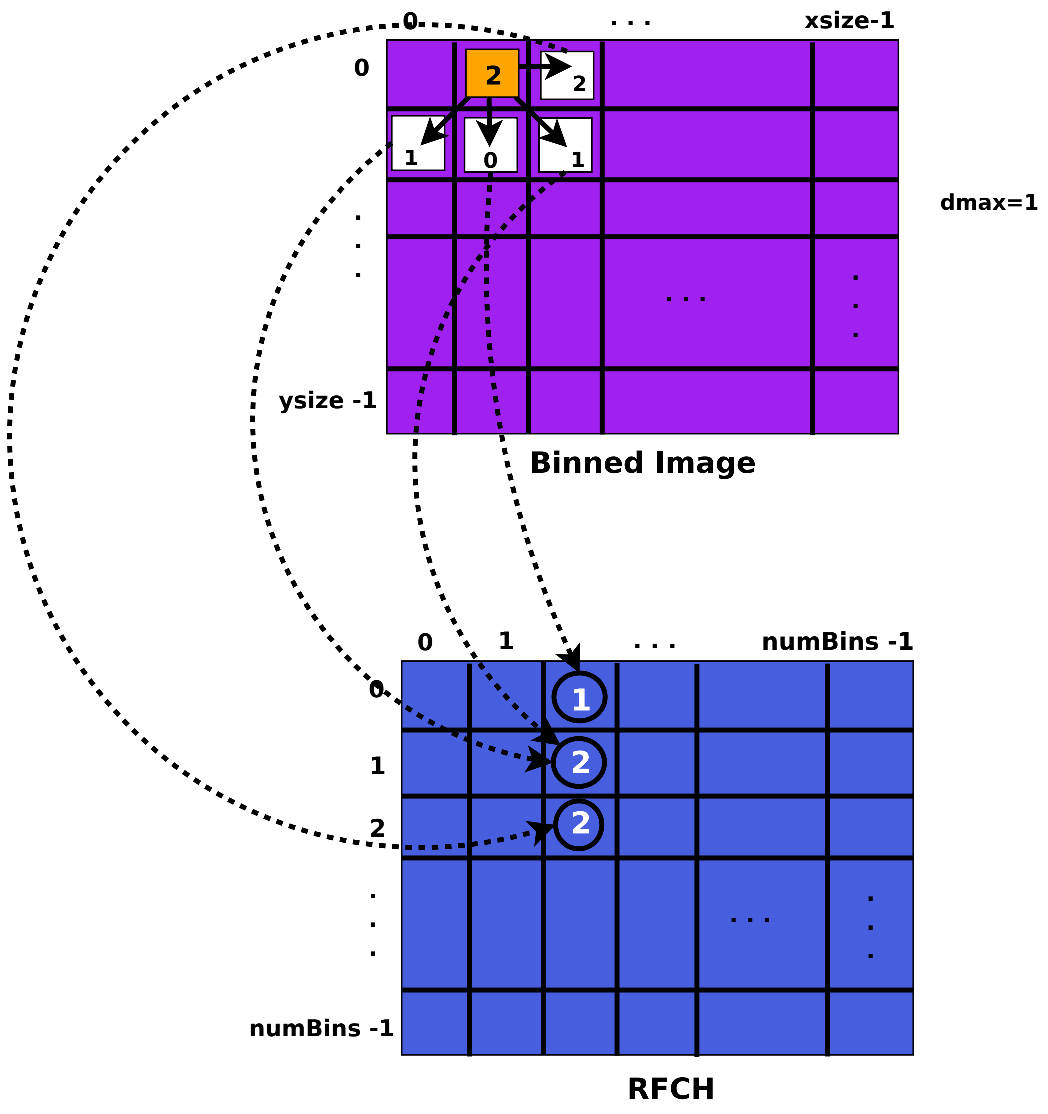
ClusterFeatures Algorithm

```
for (p=0..Image size -1) {
Cluster feature[*][p] to the nearest Cluster Center.
Store Cluster's index ('i')
binnedImage[p]=i; /* Set pixel 'p' to cluster 'i' */
}
```

Figure 5.5: Cluster Features function visualization

5.3.3 CalculateRFCH

The calculation of the Receptive Fields Cooccurrence Histograms is one more hot-spot in our collection. As the name suggests, this function calculates the cooccurrence histogram. The complexity of this function is approximately $O(nd^2)$, where n is the Image Size and d is the maximum distance (d_{max}). The figure below show how the CalculateRFCH works for $d_{max} = 1$.



CalculateRFCH Algorithm

```
for (p=0..Image size -1) {
calculate the cooccurrences of binnedImage[p] with respect of dmax
}
```

Figure 5.6: Calculate RFCH function visualization

Chapter 6

Hardware Implementation

This chapter describes the implementation of the previously identified hot-spots in hardware and more specifically on FPGA. For the implementation we have used the Xilinx Virtex-6 XC6VLX240T-1FFG1156 FPGA, which resides on a Xilinx ML605 Evaluation Board [29].

6.1 Calculate Clusters HW Implementation

6.1.1 Module Overview

As we said in previous chapters, the CalculateClusters function is responsible for the clustering of the features array, and it is performing an iterative version of the K-Means algorithm, which works in 3 phases. Phase 1 and 2 are the calculation step and phase 3 is the cluster centers update step.

The CalculateClusters Hardware module is a multi-core hardware module in which each core is performing the clustering on separate image regions (phase 1). Each core is equipped with private Data Memories (block RAMs), which are needed to achieve parallelism. When the process of the whole image finishes, each core's results are combined with each other to form a final result (phase 2). Then the update of cluster centers occurs (phase 3). The mentioned procedure is controlled by an Finite State Machine (FSM) and is repeated until convergence is reached. The module has as input the feature array and as output the clusterPoint array.

In brief we process image slices of size 640x32. This images slices contain image features forming an (feature) array 640x32x7. This means that each pixel has 7 features. The data we load into the chip is a feature array slice. We split this slice to blocks of 640x2x7 and we process each block with a separate processing unit working in parallel.

The module was designed not to work on the whole image data, but instead with sliced parts of the image one after the other. This means that the image is off-chip, and image regions are loaded into the chip when needed. Our design decision comes from the fact that in order to achieve more flexibility, keep the block RAM utilization as low as possible and be able to work on arbitrary image sizes the image data must remain off-chip.

The Module's free parameters are:

- Image xsize
- Image ysize
- Number of Cluster Centers (N)
- Number of Features (f)

The input data of the module is the feature array which is an array f x $ImageSize$ (1 byte width), where f is the number of features. As seen on the following table, in order to use the algorithm with image size 640x480 and also use 7 features, which is needed for a reasonable detection rate, a total of 2.0 MB block ram is required on the FPGA only for storing the feature array. If now, we decided to use 10 features we need approximately 1 more megabyte of block RAM on chip. The switch from 7 features to 10 features on a high definition image costs up to 2.5 MB.

An other thing to mention is that by loading all the feature array on the chip, we are not only limited by the image size and the number of features that we can use but also we target our architecture to only high-end FPGA devices.

So in order to support different combinations of image sizes and features and also target our design to low-end devices,we keep the feature array off chip, and we load slices one after the other in order to process all the array. This of course has the cost of the extra implementation effort that is needed and also makes our design I/O intensive.

Image Size (px)	**Number of Features**	**Storage needed on-chip (MB)**
640 x 480	3	0.87
640 x 480	7	2.0
640 x 480	10	2.9
1024 x 768	3	2.25
1024 x 768	7	5.25
1024 x 768	10	7.5
1280 x 720 (HD)	3	2.6
1280 x 720 (HD)	7	6.1
1280 x 720 (HD)	10	8.7

Table 6.1: Storage needed on-chip for different number of features and image sizes

6.1.2 Feature Array decomposition

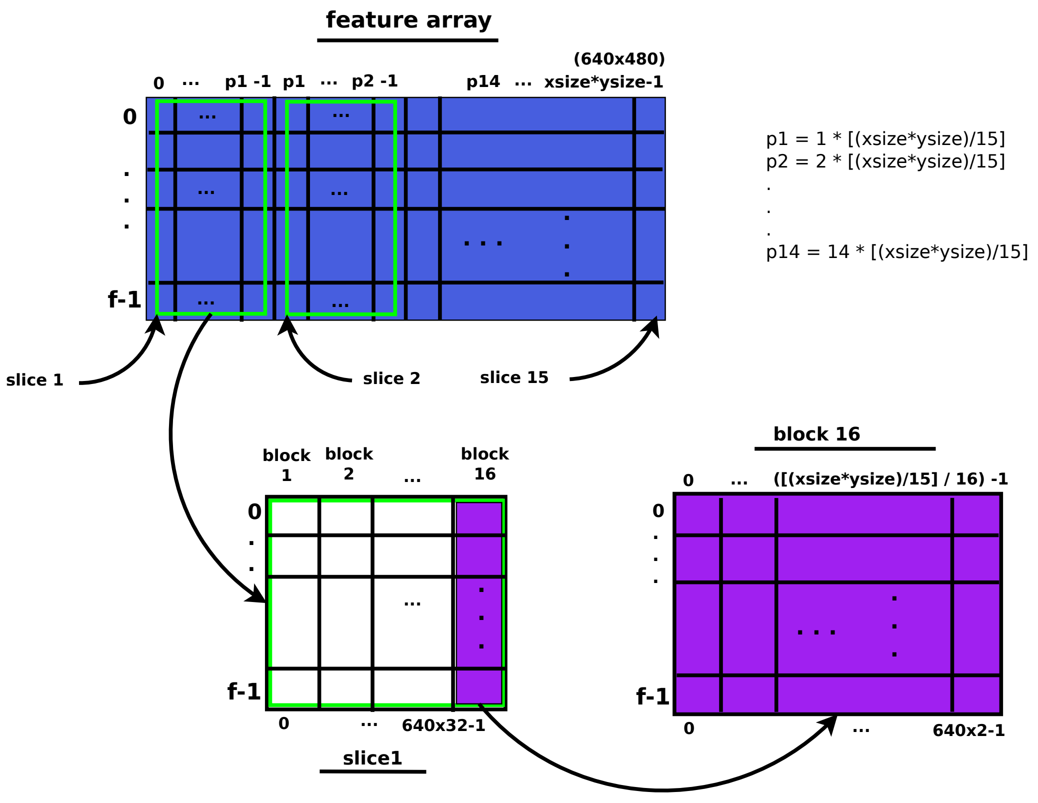
For our implementation we have used images with size 640 x 480 and also we have evaluated our design with 7 features. This means that our feature array has size equal to 640x480x7. In order to keep the feature array off-chip we follow the procedure below:

1. First we split the feature array into 15 slices with size (640x480x7)/15 = 0.13 MB. This is the data that goes on-chip, i.e. in order to process the whole feature array we have to process 15 slices. So by looking at the Table 5.1 we can see that we reduced the on-chip memory requirement by 93.5%. (The decomposition is performed in software, running on a soft-core CPU which is attached to our system, see Chapter 8).

2. Secondly we split further the on-chip slice into 16 blocks with size (640x480x7)/240 = 8.75 KB and then we pass each block to a processing core, i.e we utilize 16 processing cores in order to process one slice. The processing cores are working in parallel, and each one saves its results to private BRAMs.

The decomposition we described here is only limited by the size of block RAM that we have available on-chip and we want to utilize. Moreover the number of processing cores depends on the number of blocks we use. For example if we split the slice to 32 blocks, we can utilize 32 processing cores. In general the more block ram we have available the more processing cores we can utilize.

The figure below shows the data decomposition.



feature array
0 ... p1 -1 p1 ... p2 -1 p14 ... xsize*ysize-1
(640x480)
0
f-1
slice 1
slice 2
slice 15
p1 = 1 * [(xsize*ysize)/15]
p2 = 2 * [(xsize*ysize)/15]
p14 = 14 * [(xsize*ysize)/15]
block 1
block 2
block 16
0 ... 640x32-1
slice1
block 16
0 ... ([(xsize*ysize)/15] / 16) -1
0 ... 640x2-1

Figure 6.1: Data decomposition

6.1.3 Hardware Architecture

In the figure below we can see the block diagram of the CalculateCluster HW module.

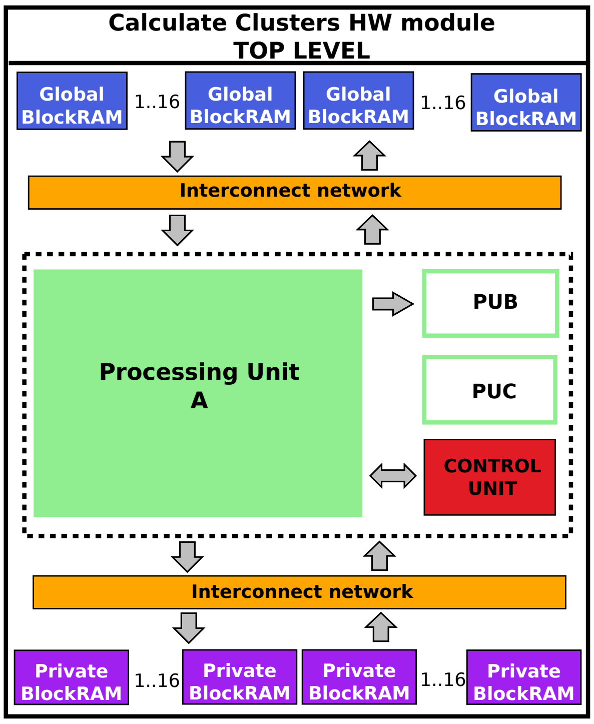


Figure 6.2: Calculate Clusters top level

In the figure above, Processing Unit A (PUA) is responsible for calculating the cluster centers and it utilizes 16 cores. Each core is able to process information that equals to a small image with size 640 x 2 , which means that the feature array of each core is 640 x 2 x 7 (8 bit) (feature array block). With the current configuration can process 16 feature image blocks (640 x 2 x 7) simultaneously.

Also in same figure the global block RAM indicates that the block RAM is accessible from the software, while the private block RAM is not. Each core, stores the intermediate results to private block RAMs. When the processing is done, PUB is responsible to sum all the intermediate results produced by the 16 cores. After PUB completion, it signals PUC which is responsible to update the cluster centers.

The Control Unit, is basically an Finite State Machine (FSM) which controls all the mentioned processing units and checks if the whole clustering process finishes, and when it does it sets the done signal to TRUE.

Below we can see a more detailed figure of the module's architecture.

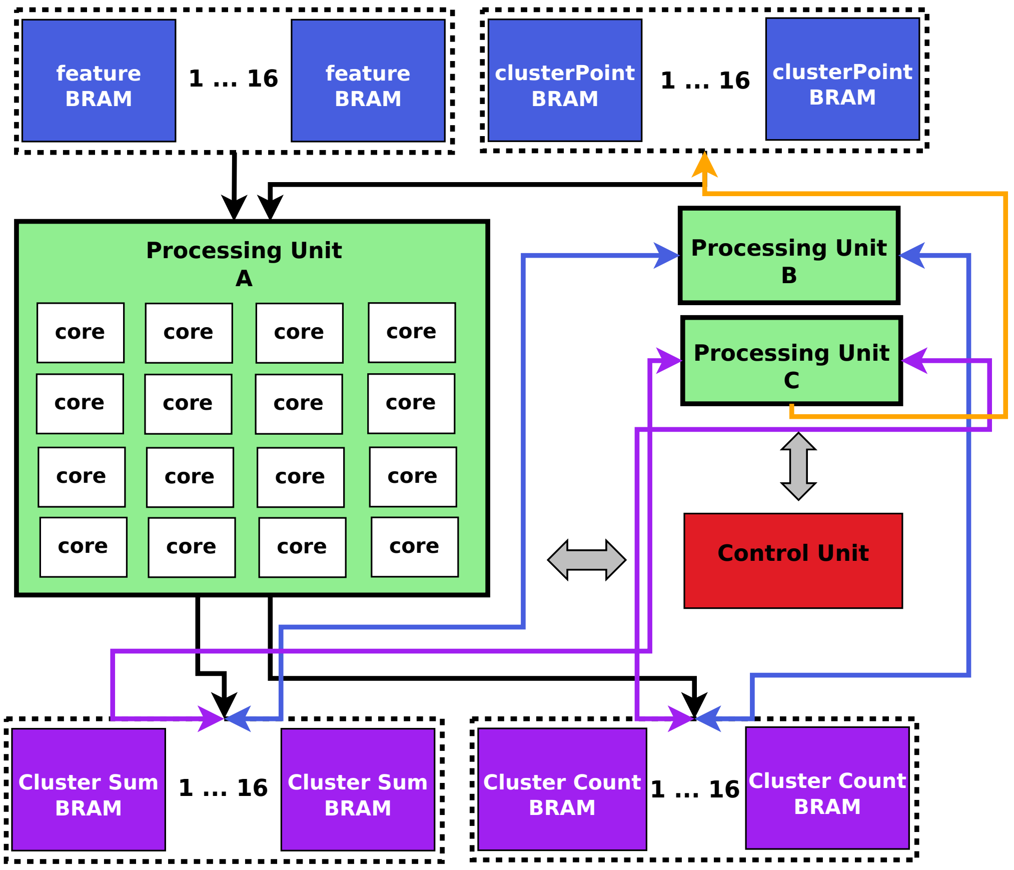


Figure 6.3: Calculate Clusters top level in more detail

Each core of the processing unit A performs step 1 and 2 as seen on Figure 5.4. Each core accesses 4 block RAMs, the two are global and the other two are private. Specifically the I/O of each core is as follows:

- Read the Feature block RAM (640 x 2 x 7 - 8 bit)
- Read the ClusterPoint block RAM (80 x 7 - 8 bit)
- Read/Write the ClusterSum block RAM (80 x 7 - 25 bit)
- Read/Write the ClusterCount block RAM (80 - 20 bit)

We have implemented the core as an Finite State Machine (FSM), which can be described by the following figure.

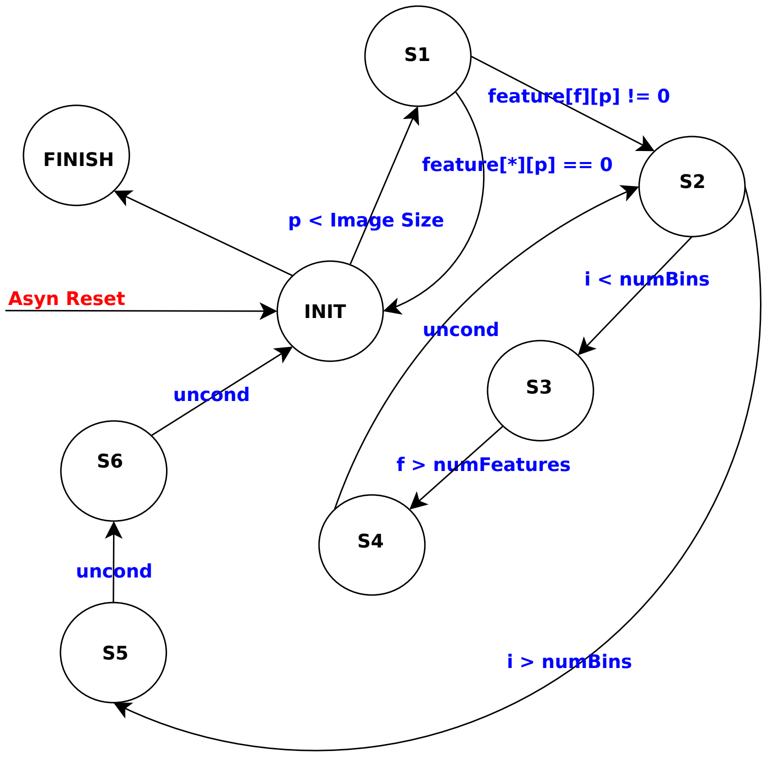


Figure 6.4: CalculateClusters Core FSM

More specifically the process performs as follows :

- On reset (asynchronous) the FSM starts at state INIT. In this state we check if there are pixels to process, and if there are we go to state S1 with the current pixel (p) Otherwise all the image has been processed and we go to the state FINISH where we stay until the reset signal is asserted.

- In State S1 we check if the specific pixel (p) has non zero features. Basically we scan the feature array from feature[0][p] to feature[f-1][p] where f is the number of features. If we find a non zero feature we go to state S2, otherwise we go back to INIT in order to check the next pixel.

- In State S2 starts the actual clustering procedure. From state S2 to state S4 what we do is that we try to find the appropriate cluster for feature[*][p]. In S2 we check if we scanned all cluster centers denoted as numBins in the figure above. If all cluster centers (i) have been checked then we have found the appropriate cluster for feature[*][p] and we go to State S5, otherwise we continue with the process and we go to State S3 with the cluster center i. The appropriate cluster center for feature[*][p] is the cluster center with the minimum distance from the feature[*][p] where the distance is given by the squared Euclidean distance.

- In State S3 we calculate the squared Euclidean distance between the current feature (p) and the current cluster center (i). Cluster centers are stored in the ClusterPoint array. So in this state we calculate the distance between feature[*][p] and clusterPoint[*][i], where $*$

indicates all features. Specifically we calculate the below formula

$$distance = \sum_{f=0}^{F} (feature[f][p] - clusterPoint[f][i])^2 \tag{6.1}$$

where F denotes the number of features. After we calculate the distance we go to State S4.

- In State S4, we find what is the minimum distance calculated so far and we store the index of the cluster center with the minimum distance (as the following code snippet shows).

  ```
  if (dist < minDist) {
          minDist = distance;
          tmp = i;
      }
  ```

  After that we go back to State S2 to continue with the next cluster center i. We repeat the states S2, S3, S4 for all cluster centers (numBins), which in our case are 80.

- In State S5 we save the information needed in order to be able to update the cluster centers later. We implemented the following code snippet :

  ```
  for (f = 0; f < numFeatures; f++)
          cluster_sum[f][tmp] += feature[f][p];


   cluster_count[tmp]++;
  ```

  When we do that, we go to State S6.

- In State S6 we just increment the pixel counter and we go to State INIT.

One important note considering the implementation of the core, is that in the State S3 we had to implement the following code snippet

```
for (f = 0; f < numFeatures; f++) {
        dist += (feature[f][p]-clusterPoint[f][i])*
                (feature[f][p]-clusterPoint[f][i]);
}
```

which is basically the equation (6.1).

As we can see from the above code we have a multiplication and immediately after that we have an addition, which actually is an accumulation. In other words this is a multiply-accumulate (MAC) operation and can be efficiently performed by a Digital Signal Processor (DSP). Our platform, Virtex-6, has built in DSP slices, namely Xilinx XtremeDSP slice which can be used to implement DSP functions.

We have implemented the mentioned MAC instruction using a Xilinx XtremDSP slice as the figure below shows.

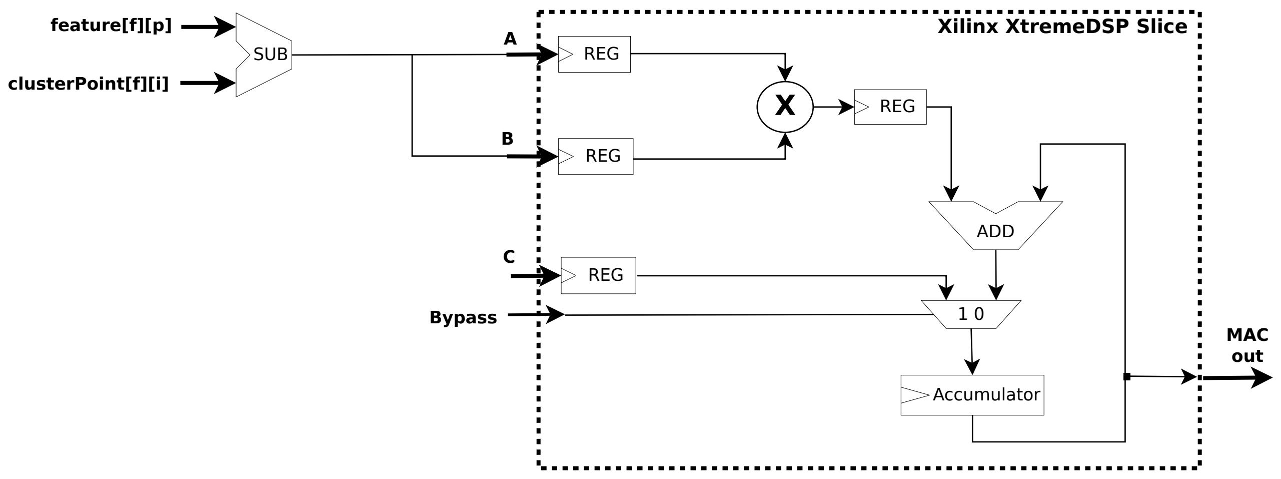


Figure 6.5: MAC instruction implementation using DSP slice

The specific implementation that uses the DSP slices is fully pipelined with 3 clock cycles latency. Each core of the Processing Unit A requires a DSP slice to carry out the above operation. In our case we have implemented 16 cores so we need 16 DSP slices. Of course we are far away from the 768 available DSP slices offered in XC6VLX240T which we are using.

When the Processing Unit A finishes, we have successfully processed an image slice 640x32. Then we repeat the exact process for the next image slice, until we process all the image (15 slices). When all 15 slices have been processed, each cores results must be added together to form the final result. This is performed by Processing Unit B, which it has as basic element a tree of adders. This unit takes as input the output of 16 different block RAMs and adds the 16 contents together.

In the next page we can see the PUB basic element. The module is fully pipelined in order to achieve high clock rate and has a latency of 4. The whole process is finished when we add all the contents of the block RAMs. We utilize 2 of the below element, which we use one for the ClusterCount block RAM and the other for the ClusterSum Block RAM.

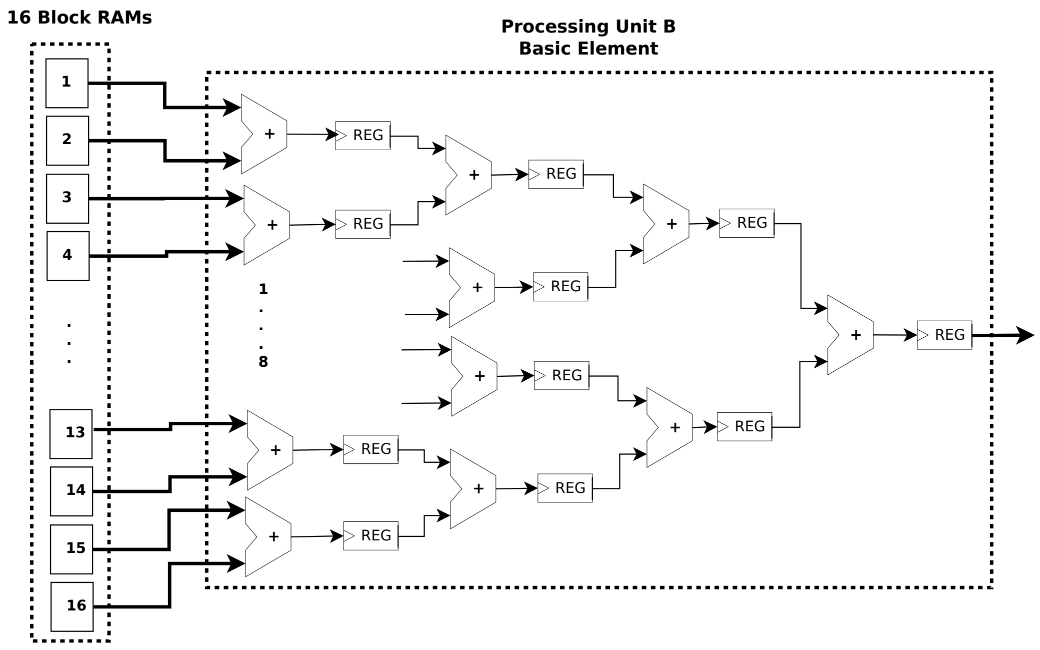


Figure 6.6: Processing Unit B Tree of adders

After Processing Unit B is done with the processing, sends the enable signal to the Processing Unit C which is responsible for the update of the clusters. Again here the module is implemented as an FSM, which can be described by the figure below.

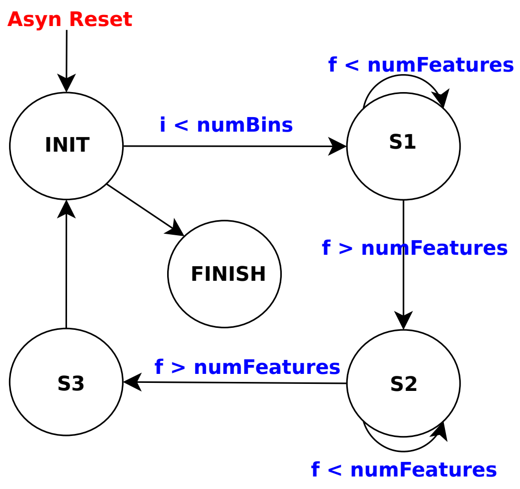


Figure 6.7: Processing Unit C

The process finishes when all cluster centers are updated. The steps performed are as below:

- The INIT state is the start-up state. Here we check if we processed all cluster centers i and if not we go to state S1 otherwise we go to FINISH.
- In state S1 the only thing we do is that we save to a temporary buffer the values of cluster center i, which in our case cluster centers are described by a vector 7 x 1, because we use 7 features. This buffer has size 7 bytes and it is implemented as distributed memory. This

step is necessary to be done in order to see if there is a change in this cluster after we update it. When we finish, we go to state S2.

- In this state we perform the actual update of the clusters. This step uses a divider generated with Xilinx CoreGen to perform the following division for all features:

$$ClusterPoint[f][i] = \frac{clusterSum[f][i]}{clusterCount[i]} \quad \forall \;\; f \tag{6.2}$$

  In the algorithm's code which is written in C++ clusterPoint is declared as unsigned char while clusterSum and clusterCount are declared as float. However, despite the fact that clusterSum an clusterCount are declared as float they actually contain values of the form "$d_1d_2d_3.00$". In the algorithm when this division is performed a cast is made to unsigned char. In C/C++ this has the effect to take truncated the result of the division. Because this is the case in the hardware we do not have a precision accuracy problem because we also truncate the result.

- In State S3 check if there is a change between the new and the old cluster center, and we store this information. Then we increase the cluster counter i and we go to state INIT in order to process the next cluster.

The control Unit of CalculateClusters Unit is responsible to coordinate the 3 processing Units and also check for the convergence of the algorithm. The figure below shows the control unit.

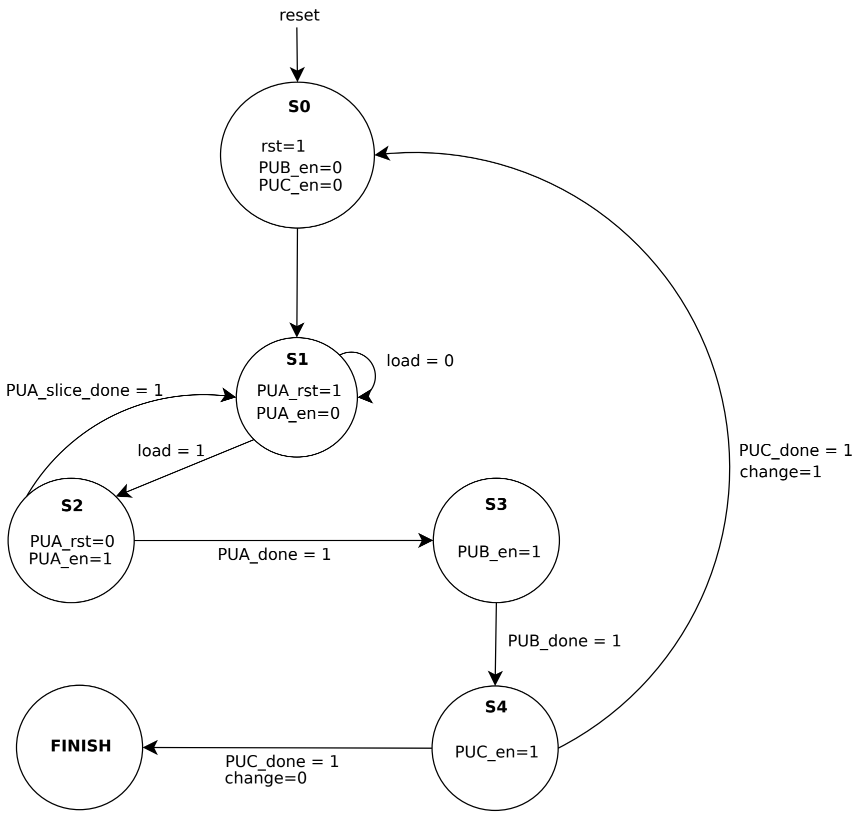


Figure 6.8: CalculateClusters Control Unit

- In the state S1 we wait for the external signal load, which indicates that the data has been transferred into the modules block RAM. When this happens we go to state S2 when we start the PUA multicore unit.
- In state S2 we wait to finish a block and when we do we go back to state S1 in order to load the next block. When all 15 blocks are processed we issue the PUA_done signal and we go to state S3.
- In state S3 we start the PUB unit in order to sum the individual results of each processors as we explained previously.
- In state S4 we start the PUC and after the processing is completed we check if there is a change in the cluster centers and we go accordingly to the appropriate state.

6.2 Cluster Features HW Implementation

6.2.1 Module Overview

This module implements the image quantization and it is also a 16-core module. As input this module takes the feature array and the clusterPoint array. Concerning the data decomposition used in this module is the same as before. Everything we said about the feature array decomposition and the clusterPoint array in the previous section also applies here.

In brief the operation of this module is as follows:

A slice of the feature array (640 x 32 x 7) is loaded into the chip and more specifically into the 16 distinct feature block RAMs. Each feature block RAM can hold a block 640 x 2 x 7. Then the clusterPoint array (7 x 80) is loaded into the 16 distinct clusterPoint block RAMs (7 x 80).

After the load is completed, the processing starts with each core to perform the quantization on an image block 640 x 2, i.e the first core quantize the image pixels 0 to 1279, the second core quantize the pixels 1280 to 2559... and so on. Again we process each image block simultaneously utilizing 16 cores.

Each core also has associated a block RAM for storing the results. This block RAM is the binnedImage block RAM which is 640 x 2 and each core has one of its own. When all the cores complete the processing, an image slice has been quantized and the results reside in the 16 binnedImage block RAMs. After that this result is upload to the DRAM and a new feature array slice is downloaded from the DRAM into the FPGA, in order to continue with the next slice. The whole process finishes when we process all the image, that is 15 slices.

The Module's free parameters are:

- Image xsize
- Image ysize
- Number of Cluster Centers (N)
- Number of Features (f)

6.2.2 Hardware Architecture

The image below shows the hardware architecture of the module.

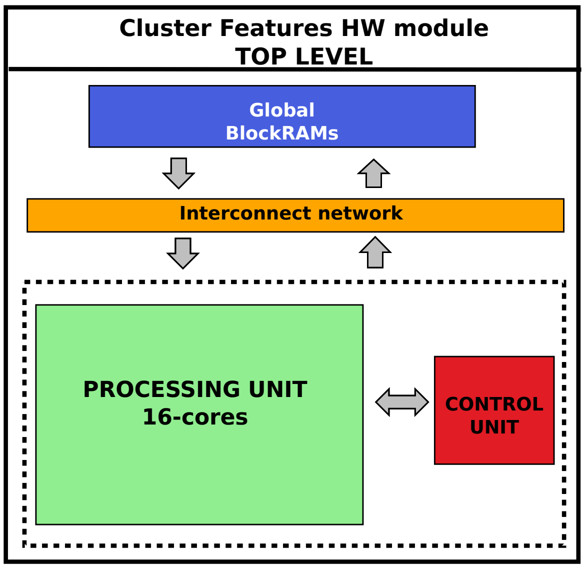


Figure 6.9: Cluster Features top level

A more detailed block diagram of the ClusterFeatures HW toplevel follows.

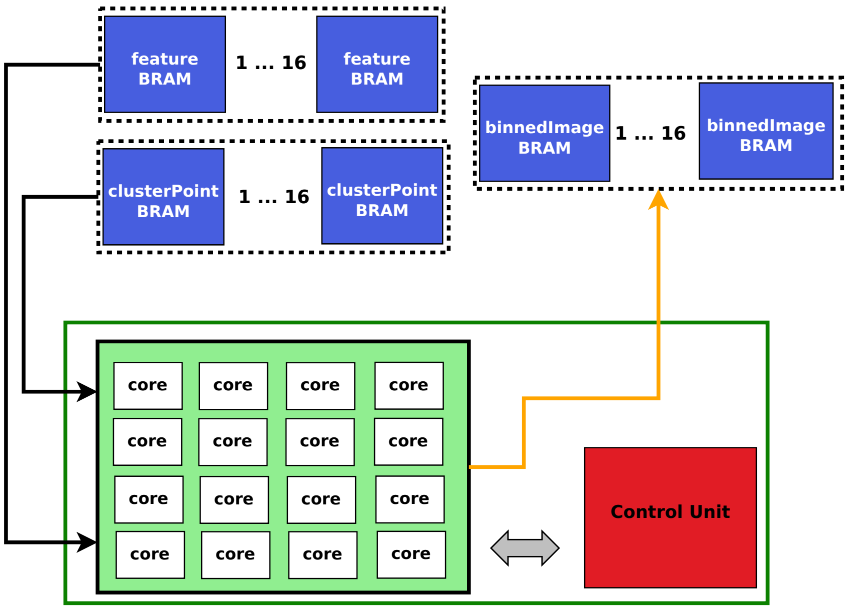


Figure 6.10: Cluster Features top level in more detail

Each core performs the same MAC operation as the CalculateClusters core, so as before we used a DSP slice in order to carry out the specific operation. In total we need again 16 DSP slices for all the module. Each core implements an FSM which is explained in the next page.

The figure below describes the operation of the ClusterFeatures FSM.

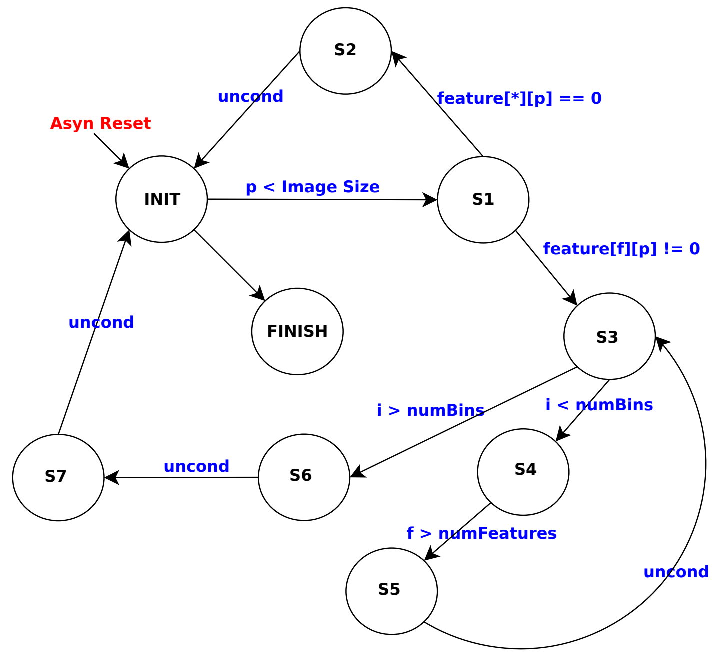


Figure 6.11: ClusterFeature core

The processing follows the below steps :

- The INIT state checks for completion. In other words check if we processed all the image pixels p and if not go to state S1 otherwise we go to the FINISH state where we assert the done signal.
- In state S1 we read the features of pixel p from the feature block RAM and if at least one feature exists then we go to state S3 else we go to state S2.
- In state S2, we came because we did not find any features for pixel p, so we set the pixel p of binnedImage to the value -1. This indicates the null entry. Then we increment the pixel counter and we go back to state INIT
- In state S3 we check if we processed all cluster centers i for pixel p and if we did we go to state S6, otherwise we go to state S4.
- In state S4 we implement the below code snippet, which is the same as CalculateClusters:

```
for (f = 0; f < numFeatures; f++) {
        dist += (feature[f][p]-clusterPoint[f][i])*
                (feature[f][p]-clusterPoint[f][i]);
}
```

  This is where we use the DSP slice, as described in the previous section.

- In state S5 we keep track of the cluster center i which corresponds to the minimum distance with the features of pixel p and we set the binnedImage[p]=i. Then we increment the cluster counter and we go back to state S3.
- In state S6 we check the case in which the minimum distance calculated above is very big which means that the specific feature of pixel p actually does not belong any cluster i, despite the fact that we associate it with a cluster center, so we overwrite it with the null value (binnedImage[p]=-1).
- Finally in state S7 we increment the pixel counter and we go to state INIT.

The control unit of the ClusterFeatures module is quite simple. As the figure below shows the operation of this unit is to AND the 16 done signals of all processing cores.

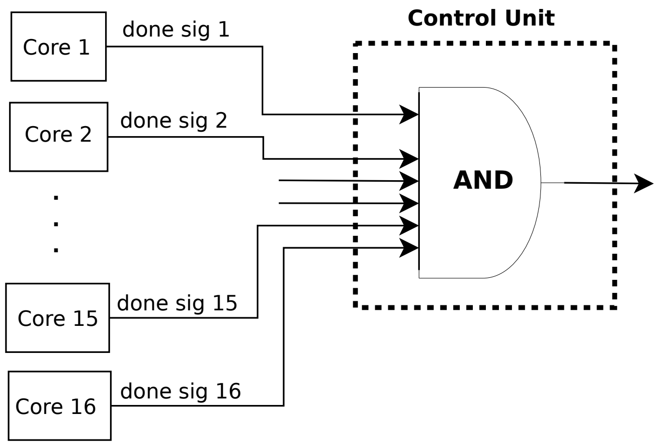


Figure 6.12: ClusterFeatures Control Unit

6.3 Calculate RFCH HW Implementation

6.3.1 Module Overview

This module implements in hardware the calculation of the receptive fields cooccurrence histogram. This module utilizes 8 processing cores. Each core is responsible to calculate the cooccurrences of an binnedImage block that is 640x4. This means that the 8 cores together can calculate the cooccurrences of an binnedImage slice equal to 640x32. The cooccurrenses are calculated with respect to d_{max}=4, although if we utilize more block RAM we can support d_{max} over 4. Each processing core maintains a block RAM for the output where the RFCH is stored. This block RAM is 80x80 - 11 bits. In order to calculate the RFCH of the whole binnedImage 640x480 we have to process 15 binnedImage slices 640x32. When we process all the image slices then the 8 RFCH BRAMs are summed to form the final RFCH for the image and then the result is carried out.

6.3.2 BinnedImage Array Decomposition

The RFCH is calculated on the Binned Image data. The binnedImage array which hold the Binned Image data is an array 640 x 480 and as we said previously binnedImage is the quantize version of the image. In order to calculate the RFCH of binnedImage, for each binnedImage slice 640 x 32 we form a bigger binnedImage slice which is 640 x 64.

This new slice holds an extra 4 lines of the image data for each core, in order to serve the d_{max}=4 condition. As shown in the following figure for every binnedImage block (640x2) we need the following block too and for every 32 lines - slice , we need the following 4 lines.

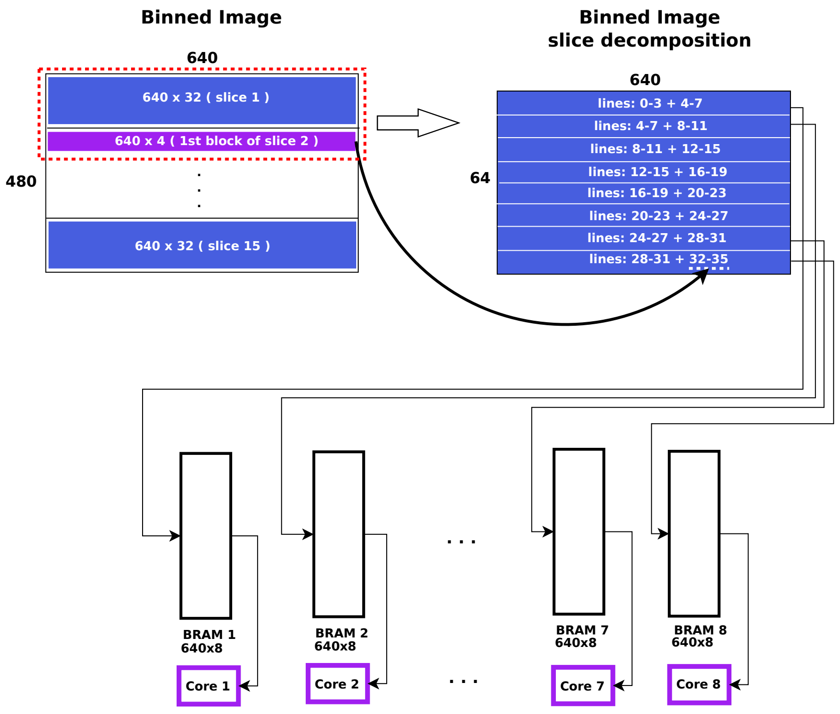


Figure 6.13: Binned Image decomposition

We load the 8 blocks 640 x 8 to the 8 block RAMs of the module, and so each processor can calculate the RFCH of a binnedImage part 640 x 4 with d_{max} up to 4. For each binnedImage slice 640 x 32 we need to load also a block (640 x 4) from the next slice as the figure shows in order to be able to look ahead for cooccurrences in distance=4.

6.3.3 Hardware Architecture

The below figure shows the block diagram of the module.

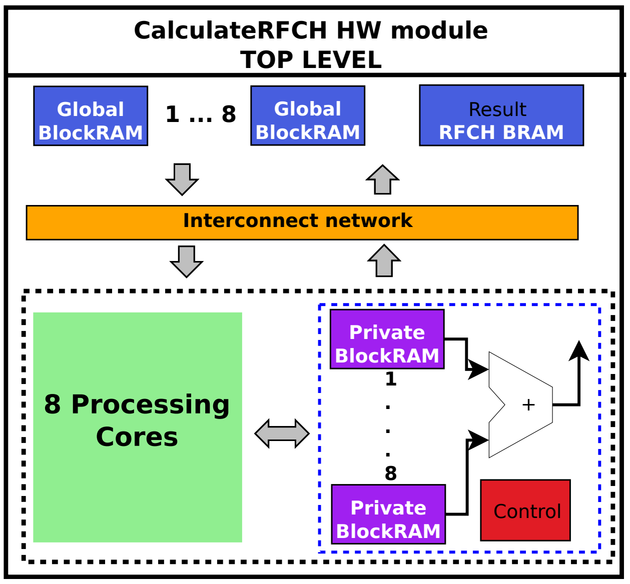


Figure 6.14: Calculate RFCH top level

The figure above shows the block diagram of the CalculateRFCH HW module. Each processing core reads and writes from its own block RAMs. When all the binned image is processed the control is responsible to start the summation of the results which are stored to the Result RFCH RRAM as shown above

The operation of the core can be explained as below:

```
for $y = 0$ ... 3 do
  for $x = 0$ ... 639 do
    {Search for cooccurrences in the neighborhood of (x,y)}
  end for
end for
```

We have implemented each core as an FSM which executes the the above code. Each time we load a different binnedImage slice to the block RAMs and so we can process all the binned image.

6.4 Device Utilization

Below we can see the device utilization on Virtex-6 VLX240T. As we can see by leaving the image off-chip we reduced the block RAM usage. Also there is a lot of space to utilize more processing units for better performance.

Xilinx Virtex-6 VLX240T-1FFG1156				
Module	**Slice Registers**	**Slice LUT**	**Block RAM**	**DSP Slices**
Calculate Clusters	14333/301440 (4%)	16385/150720 (10%)	34/416 (8%)	16/768 (2%)
Cluster Features	4176/301440 (1%)	5858/150720 (3%)	56/416 (13%)	16/768 (2%)
Calculate RFCH	1056/301440 (0%)	2819/150720 (2%)	16/416 (3%)	0/768 (0%)
Total	19565/301440 (6%)	25062/150720 (16%)	106/416 (25%)	32/768 (4%)

Table 6.2: Device Utilization

6.5 Hardware verification

For all the three hardware modules that were implemented we followed the below verification process in order to check their functionality :

1. **Complete Behavioral Simulation:** What we did in this step is that, we first downloaded images from the CODID data-set [28], and then we rescale them to 640 x 480 in order to fit to our needs. Then we ran the algorithm a specific configuration (i.e Number of features=7, Number of clusters=80) and we wrote down the results. After that we ran the same test on our hardware and we verified that the results match.More specifically in order to verify the functionality of the CalculateCluster module which takes as input one slice (640 x 32) of the feature array, we first run the algorithm for an object. Then we took the software calculated feature array and we sliced it into 15 pieces and after that we created 16 blocks from each piece. We wrote the information of these 16 blocks to COE files and then we loaded the 16 COE files into the 16 block RAMs of the module using CORGEN. Then the simulation performed and the results verified. Equivalent procedure was performed for the rest of the modules.

2. **Place and Route:** After the behavioral simulation we performed place and route to our modules, which completed successfully.

3. **Post Place and Route Simulation (i.e Timing Simulation):** In this step we produced post place and route simulation models for our modules and we performed simulation but with a data-set smaller than the one we used in step 1. We verified that results are correct.

4. **Testing on the board after the download:** In this step we have downloaded the designs to the board (ML 605) and we verified their correct operation.

Chapter 7

Hardware Performance Evaluation

In this Chapter we show how the custom hardware performs and we compare it with the software application.
The hardware performance was tested with the following hardware configuration:

- Number of Calculate Clusters cores : 16
- Number of Cluster Features cores : 16
- Number of Calculate RFCH cores : 8
- Image Size : 640 x 480
- Number of Clusters(N) : 80
- Number of features(f) : 7
- Maximum distance (d_{max}) : 4

The modules are able to achieve the maximum (synthesis) clock rate of 300 MHz on the Xilinx Virtex-6 VLX240T -1. The table below shows how many clock cycles are needed in order to process a common CODID [28] image and object.

Module	**Clock Cycles**
Calculate Clusters (1 object)	5941290
Cluster Features (1 object)	6130470
Cluster Features (1 image)	32448045
Calculate RFCH (1 object)	2584500

Table 7.1: Clock Cycles needed by the Custom Hardware

According to the table 7.1 we converted the above cycles to time using two frequencies at 200MHz and at 300MHz which is the maximum achievable frequency. The times are listed in the tables below. We also list the times at 200MHz in order to have a better picture of what is happening, and also because some low-end devices might not be able to hit the clock rate of 300MHz so this is a good estimate, to know how the hardware performs on a lower frequency too.

Module	Time @ 200 MHz (s)	Time @ 300 MHz (s)
Calculate Clusters (1 object)	0.0297	0.0198
Cluster Features (1 object)	0.0307	0.0204
Cluster Features (1 image)	0.1622	0.1082
Calculate RFCH (1 object)	0.0129	0.0086

Table 7.2: Time needed by the hardware for two frequencies

The following sections show the achievable speedup over the software. The software times taken with a ULV CPU @ 1.3GHz.

7.1 Speedup at 200 MHz

As we can see from the table below at 200 MHz, Calculate Clusters is over 10 times faster compared to the software while Cluster Feature is about 16X and 11X faster for 1 object and 1 image respectively. On the other hand Calculate RFCH[1] is almost 4X over the software version

Module	Software Time (s)	Hardware Time (s)	Speedup
Calculate Clusters (1 object)	0.3200	0.0297	10.7
Cluster Features (1 object)	0.5000	0.0307	16.2
Cluster Features (1 image)	1.8100	0.1622	11.1
Calculate RFCH (1 object)	0.0500	0.0129	3.8

Table 7.3: Speedup at 200 MHz

7.2 Speedup at 300 MHz

In the case of 300MHz the speedup gained is much more as we see in the table below.

Module	Software Time (s)	Hardware Time (s)	Speedup
Calculate Clusters (1 object)	0.3200	0.0198	16.2
Cluster Features (1 object)	0.5000	0.0204	24.5
Cluster Features (1 image)	1.8100	0.1082	16.7
Calculate RFCH (1 object)	0.0500	0.0086	5.8

Table 7.4: Speedup at 300 MHz

[1]This module utilizes 8-cores while all the others are 16-core modules

7.3 Speedup at 350 MHz

Our hardware can achieve a clock rate of up to 350MHz on the Virtex 6 - Speedgrade - 2, giving additional speedup over the software as the table below shows.

Module	Software Time (s)	Hardware Time (s)	Speedup
Calculate Clusters (1 object)	0.3200	0.0170	18.8
Cluster Features (1 object)	0.5000	0.0175	28.5
Cluster Features (1 image)	1.8100	0.0927	19.5
Calculate RFCH (1 object)	0.0500	0.0074	6.7

Table 7.5: Speedup at 350 MHz

7.4 Overall Speedup

According to section 5.2.5 and using Amdahl's equation we calculated the overall speedup. The results are listed to the following table:

Configuration	@ 200 MHz	@ 300 MHz	@ 350 MHz
1 object - 1 scene	7.2	9.5	10.5
3 objects - 1 scene	8.4	10.3	11.5
10 objects - 1 scene	9.4	13.9	15.9
10 objects - 10 scenes	8.6	13.1	15.1

Table 7.6: Overall Speedup

As we see from the above table the overall speedup is roughly 10X at 200MHz, about 14X at 300MHz and upto 16X at 350MHz. The slight decrease that we see when we go from 1 scene to 10 scenes is because the time consumed by the Cluster Features and the Calculate RFCH is growing while the time consumed by the Calculate Cluster stays stable. This is because Calculate Clusters is only for the training. When we perform the testing, only the two functions mentioned are executed, and because calculate RFCH has a lower speed up because of the 8-core implementation we see this decrease.

7.5 Conclusion

From the results we see that by parallelizing the above functions we can achieve a significant speedup over the sequential software version. More over because the algorithm spends a lot of its time executing the 3 functions, further speedup can be achieved by utilizing more processing units, since also the device utilization is very low.

Chapter 8

Embedded System Design

After the hardware implementation of the algorithm's hot-spots, we studied the scenario of designing an embedded system on FPGA for the algorithm and we also have prototyped such a system. In this chapter we describe the embedded system that we have prototyped. The time that the author writes these lines despite all his efforts up to this time, the system does not meet the target's platform requirements, for reasons explained below, and in order to do so additional attention is required. However, in this first attempt we succeeded a proof of concept.

8.1 System Overview

Our embedded system was prototyped on the Xilinx ML605 Evaluation board [29]. The specific board utilizes the Xilinx Virtex-6 VLX240T-1FFG1156 [3].

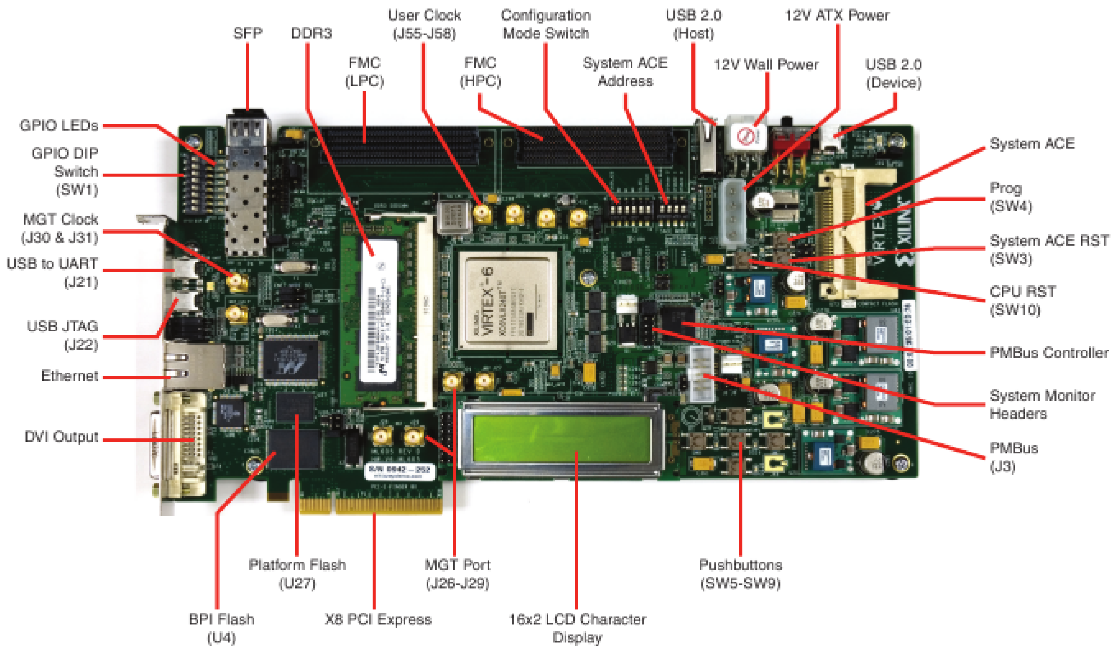


Figure 8.1: Xilinx ML605 Evaluation Board

The basic components of our system are :

- Xilinx System ACE CompactFlash (CF) controller [30]
- DDR3 SODIMM [33]
- Microblaze Softcore processor [34]
- RFCH Hardware Accelerator

8.2 System Architecture

The below figure shows the block diagram of our embedded system. We analyze each component in the next page.

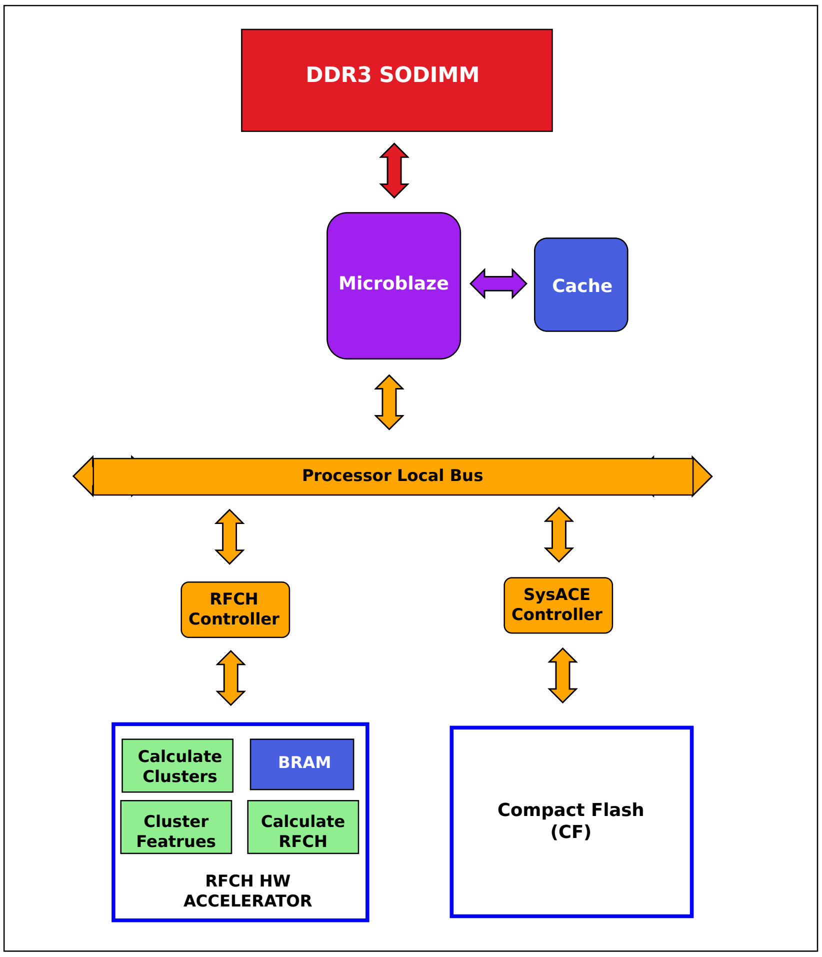


Figure 8.2: RFCH embedded system block diagram

8.2.1 Compact Flash

We use the compact flash to store all the training and testing images. This is the standard input method to the system. The algorithm is using the Portable Pixmap (PPM) format for the image encoding, which is defined in the Netpbm open source library [35].

The PPM format comes in two forms, the one is the ASCII PPM (P3 PPM) format and the other is the binary PPM (P6 PPM). The RFCH algorithm is normally using P6 PPM for the images, however the Xilinx's sysace_fread() function [31], can only work with ASCII files, that is, it can not use the “b” flag that the C/C++ fread() implementation supports.

In order to overcome this problem we developed a conversion routine that handles PPM P3 to PPM P6 conversions. We store to the compact flash P3 PPM images, we read them using the sysace_fread() function and then we convert them to P6 using the P3_to_P6() conversion routine and we pass them to the algorithm. All the above have been integrated to the embedded version of the RFCH algorithm.

8.2.2 DDR3

The system needs definitely the external DDR SDRAM in order to be able to operate. More specifically, data such as images, features and other require a significant amount of storage and the only option we have is to use the DDR.

Additionally, the memory requirements increase as the number of training objects increases along with the image size, the number of features and the number of clusters.

The figure below shows the heap requirement (heap profiling) that is needed by the algorithm when we train 3 objects (f=7, N=80, Image Size=640x480). As the figure shows we have a heap peak at 18 MB for only 3 objects.

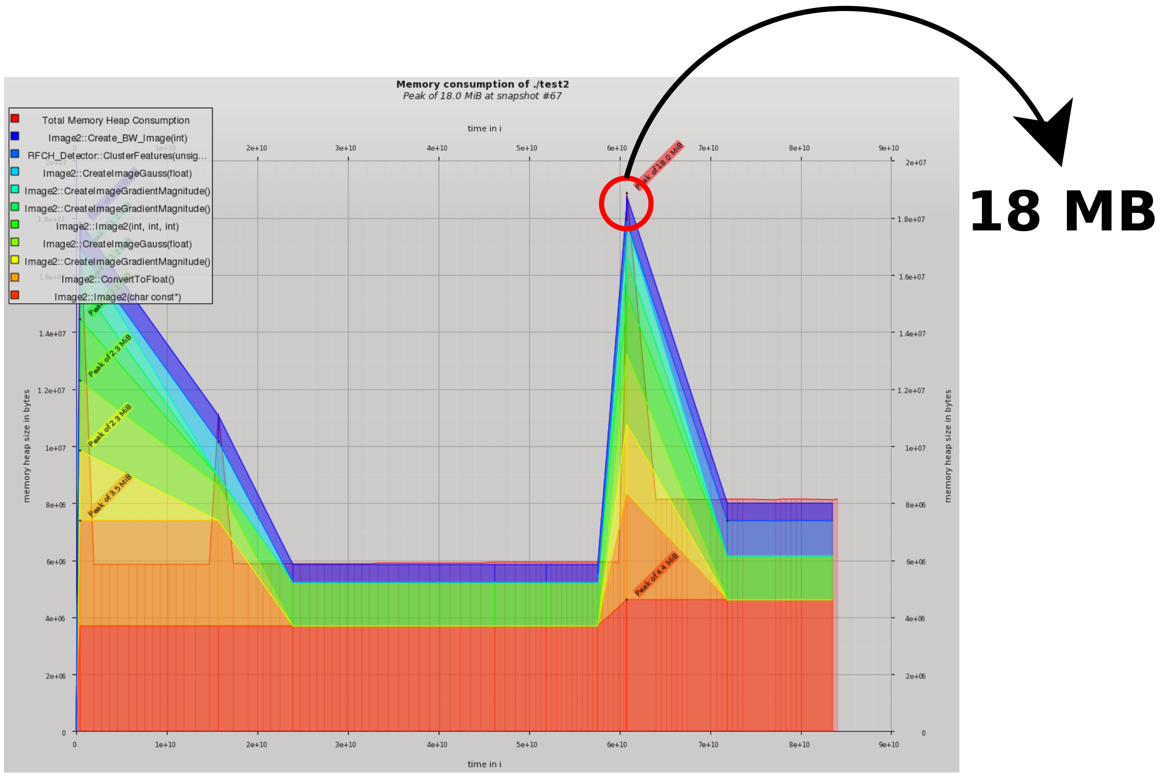


Figure 8.3: Heap requirement for 3 objects

8.2.3 Microblaze

Microblaze is the coordinator of the whole system. Its main purpose is to execute the part of the RFCH algorithm that we decided to leave in software and also it controls the system peripherals. Microblaze executes the embedded version of the RFCH algorithm, in this version the functions that were implemented in hardware have been replaced by the appropriate hardware drivers. Microblaze is also responsible for the data decomposition of the feature and binned image array as discussed on previous chapters. The decomposition is performed by software drivers. Microblaze is configured as a Standalone platform [32]. The Standalone platform is a single-threaded, simple operating system (OS) platform that provides the lowest layer of software modules used to access processor-specific functions.

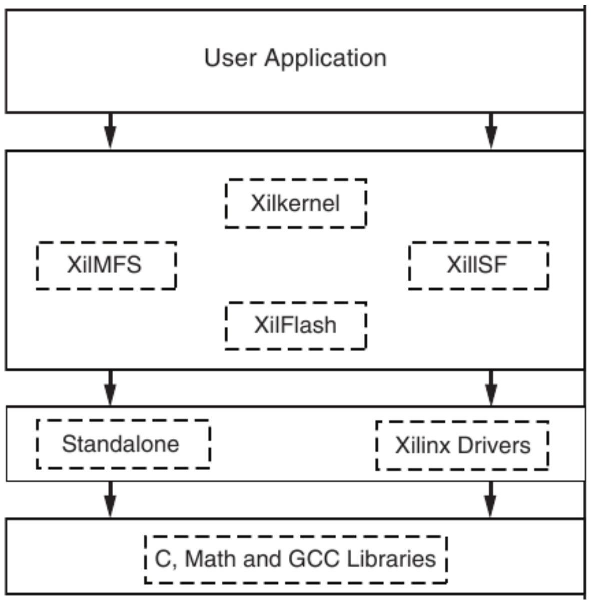


Figure 8.4: Microblaze Libraries Organization

8.2.4 RFCH Controller

The RFCH controller is the hardware between the Microblaze and the RFCH hardware accelerator, and it is attached to the PLB. This module is basically a PLB slave single [36], which provides a bi-directional interface between the RFCH hardware accelerator and the PLB. The later is possible via software accessible registers. RFCH controller was generated using an EDK PLB slave template with software accessible registers which are addressable through software.

We write the data that must be loaded on chip to these registers from software, and with a simple FSM in hardware we transfer the data further into the appropriate component (i.e block RAM) which is connected to a specific register. After the transfer is completed an acknowledgment is send to the bus indicating that the transfer is done.

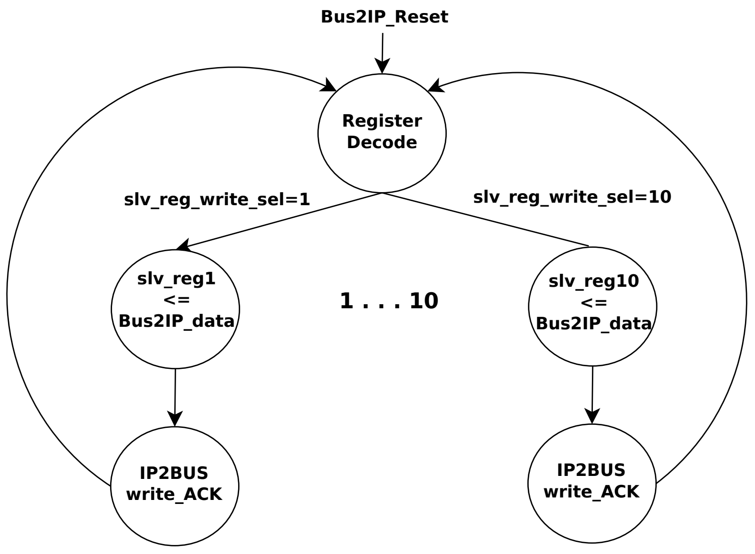


Figure 8.5: Part of the RFCH Controller responsible for the Bus2IP write

8.3 Software Drivers

The system's software drivers are developed in C++. We have developed two libraries of drivers, namely low-level drivers and high-level drivers (See Appendix).

The low-level drivers control the RFCH Controller and provide read/write access to the RFCH hardware accelerator's internal components. The high level drivers implement the appropriate functionality in order to replace the software functions CalculateClusters(), ClusterFeatures() and CalculateRFCH with the hardware accelerated ones.

8.4 System evaluation - Results

In this section we talk about how the system responses and also the difficulties we faced during the design. A profile analysis was attempted, following the instructions in [37] but unfortunately with no success.

8.4.1 Microblaze

As we mentioned earlier Microblaze among other things executes the getFeatures() software function and also reads images from the compact flash. To perform these operations Microblaze needs to access the DRAM, because all this data belongs to the heap and the heap resides on the DRAM. Additionally, the libraries needed for accessing the compact flash as well as the standalone OS libraries combined with the RFCH code, give the total microblaze code such a size, that it can not fit in the 64KB local memory (this is the memory accessed by ilmb-dlmb controller), so we have also put this code into the DRAM. Specifically the executable file (ELF) is 1.1 MB, maybe this is because we use mixed C and C++.

In order to increase Microblaze performance (for accessing the DRAM) we enabled the data and instruction caches (64K each) and also other performance features like the victim cache. However, by doing this we limited its frequency to 150 MHz. We tried to increase the frequency further but with no luck.

As of the above, the functions of the algorithm that remained in software does not perform well on the Microblaze, in this configuration. As we said earlier, unfortunately we were unable to perform a profile analysis and so we do not have exact numbers to report here, however a rough approximation is that the software on the microblaze at 150MHz is over 60 times slower from a personal computer @ 1.3GHz. The other reason that slows things down is the fact that we did not performed floating point to fixed point arithmetic conversion at this time, and so floating point operations are performed at low frequency causing extra overhead.

8.4.2 RFCH Controller

In order to train 1 object, we have to transfer approximately 2.0 MB of data from the DRAM into the FPGA. The transfer is being carried out by blocks that have size of 140 KB. So in order to perform clustering using the CalculateClusters HW module, for one object we need to transfer 15 such blocks from the DRAM to the modules block RAM.

The component that is responsible for the above process (i.e the data transfer) is the RFCH controller and the RFCH software driver. At first we tried to implement the controller using the Fast Simplex Link (FSL) that has 1 cycle latency for each data transfer, but unfortunately due to a possible bug (also reported by others in Xilinx's forum) we could not use the FSL on our platform at that time.
So we implemented the controller as PLB slave [36], but with the cost of several clock cycles for each transfer, and this becomes an other bottleneck of the system with the current implementation.Approximately with the current implementation we need about 5 ms at 150MHz to transfer 140KB into the chip. Of course as a PLB slave it is more close to the newer AMBA bus and an upgrade to support the ARM Cortex processor in the upcoming Virtex 7 it is easier to be done.

Chapter 9

Improving the performance of the embedded system

In this chapter we give our opinions and some basic guidelines on how we can solve the performance issues of the embedded system.

Regarding the embedded system the bottleneck resides in the Microblaze processor and in the I/O communication through the PLB bus using the RFCH controller. The table below shows an estimate on how the system performs compared to an Intel CPU clocked at 1.3GHz.

Embedded System			**PC**
Component	**Operation**	**Microblaze Time(s) @ 150 MHz**	**Intel CPU Time(s) @ 1.3GHz**
Microblaze	Image Gauss	3.70	0.05
Microblaze	Image Laplace	0.60	0.01
Microblaze	Gradient Magnitude	2.76	0.03
Microblaze	Read from Compact Flash[1]	2.37	N/A
RFCH controller	Transfer 140K to chip via PLB[2]	0.005	N/A

Table 9.1: Bottleneck summary

First of all the functions Image Gauss, Image Laplace and Gradient Magnitude are executed by the getFeatures function and what they do is that they extract the image features. Depending on the number of features and their combination these functions can be executed multiple times for an image. In the above table we see the time that these functions take to finish for one object (640x480 image like figure 5.2) on the Microblaze @ 150 MHz versus an Intel @ 1.3 GHz. The

[1]Read a color Image 640 x 480 ASCII PPM about 2.0 MB

[2]Using the current implementation of the RFCH Controller.

question is what can we do to make the functions faster on the embedded system.

The first thing we can do is increase the frequency. According to Xilinx, the Microblaze V8.10a configured in performance mode with the caches enabled and with an external memory controller can achieve a clock rate of 223MHz on Virtex6-3[1]. However from our hands-on experience we believe that this is still not enough for the specific computer vision application, and if there exist a better alternative we should better give it a look.

The better alternative is a hardcore processor like the PowerPC and the ARM Cortex. Specifically Xilinx's Zynq-7000 Extensible Processing Platform[39] will be featuring a Dual ARM Cortex™-A9 MPCore hardcore processor with the following characteristics:

- Clock rate up to 800 MHz.
- Enhanced with NEON Extension and Single and Double Precision Floating point unit.
- 32kB Instruction and 32kB Data L1 Cache
- Unified 512kB L2 Cache
- Out-of-order speculative issue superscalar execution pipeline.
- Dual Core processing

By executing our application on an ARM Cortex like the one mentioned above we will for sure achieve a significant improvement in performance over the Microblaze, given that we will also have a good DDR memory controller. All of our code resides in the DRAM, because it is too big to fit to the microblaze local memory. When it comes for embedded system applications we cannot always fit the application into the processors local memory and that's why a good memory controller is needed.

The second thing we can do is that we can utilize more that one Microblazes for our application. Given the fact that we kept the device utilization at low levels, we can use multiple Microblazes in order to improve the performance of the system. The algorithm is using a combination of features that extracts for each image, and this combination is configurable. For example the feature COLOR_LAPLACE executes the functions Image Gauss and then Image Laplace and the feature GRADIENT_MAGNITUDE executes the functions Image Gauss and Gradient Magnitude. The two features mentioned can be executed in parallel. In this case we can benefit from a multi-microblaze implementation. For example MB1 can execute the GRADIENT_MAGNITUDE feature and MB2 can execute the COLOR_LAPLACE feature. In the sequential version in order to extract these two features we need 10.76 seconds while in the parallel implementation using 2 Microblazes we need only 6.46 seconds which gives as 1.6X speedup.

[1]We worked with Virtex6-1

One other thing that we can do in order to increase the performance of the software running on microbaze is to go ahead and rewrite the software (i.e the functions remained in software) using the multithreding-multitasking paradigm on multiple microblazes, since we have the available resources on our device. However the implementation of a multi-microblaze system which will be consisted of over 10 Microblaze cores capable of executing any multi-threding program requires a significant engineering effort and plenty of time.

The third thing we can do regarding the software, is that we can perform a floating point to fixed point arithmetic conversion. By study the specific algorithm we can say that such a conversion will not result in any precision accuracy problems since the algorithm only uses the floating point representation (float) in order to make use of real numbers (non integers). Regarding accuracy, 2 or 3 decimal places are enough since the algorithm is mostly performing calculations with pixel values in the range [0 to 255] and additional decimal places are not significant to the result. The benefit for making such a conversion is the speedup we gain by not using floating point operations that require much more clock cycles than the integer operations. Additionally by using configurable softcore processor like Microblaze we can exclude the floating point unit (FPU) saving at the same time area resources and power consumption.

The table below shows a benchmark conducted by the authors of libfixmath[38], which is a Cross Platform Fixed Point Maths Library. The benchmark regards the execution of the arctangent function on an ARM Cortex-M0 using the C's *math.h* library and the *libfixmath* library.

Libfixmath Benchmark		
Test	**Code Size (bytes)**	**Clock Cycles**
math.h	10768	14340
fix16_atan2	4512	3772

Table 9.2: Fixed Point Arithmetic Benchmark

As we can see from the above table the code needed for this specific case has been reduced from 10.7KB to 4.5KB and about 10K clock cycles saved. Concluding, in our application we can benefit for a floating point to fixed point conversion in terms of speed, code size reduction and power consumption.

The other major bottleneck of the embedded system is the I/O. Because we do not have the image data on chip for reasons explained earlier, images slices are being transferred from the DRAM to the BRAM.

In order to improve this operation the one approach is to go back to the FSL which is quite simple and can deliver the data in 1 cycle instead of the 5-6 cycles that a single beat transfer is needed on PLB. By doing this we will going to gain a speedup over the current implementation but we are probably limiting our design to 32 bit data transfers since FSL is 32 bit wide and also we do not know if other hardcore processors like the ARM Cortex will support the FSL.

The other thing that we can do is to use the Burst Mode feature of the PLB. Specifically the PLBv46 Master Burst[40] supports fixed length burst Read/Write data transfers of upto 16 data beats on the PLB. Additionally the new AMBA bus with the AXI4 interface can be configured to use 64-bit data bus, which is something very beneficial for our I/O intensive application and also support burst mode transfers[41]. Additionally the double buffering technique can be used if we doubled our block RAMs, in order to hide the loading time in the execution time. While the hardware is working with one slice, the next slice could being transferred into the chip.

Moreover if we utilize over one Microblazes we can load different block RAMs simultaneously and so we can improve the I/O performance. Of course this will require the different Microblazes to have their own PLB bus in order to send the data in parallel.

Regarding the reading from the compact flash, as of now the Xilinx's drivers for the compact flash I/O do not support reading binary file. The specific driver can only work with ASCII files. We did not study the internals of the compact flash controller and its drivers and we can not tell if it is possible and how easy it is to write a driver that reads binary files, however we can gain a significant speedup if we can read binary files from flash.

Specifically, a binary 640 x 480 color PPM Image containing an object like figure 5.2 is about 900 KB while the ASCII version of the same file is almost 2.0 MB. In our platform we have to read from the compact flash the ASCII version of the image resulting a slower reading time. In the case that we will able to read binary files from the compact flash and with a slight modification of the algorithm, compressed images can be used to reduce further the I/O time on the compact flash.

We believe that with a little more research and considering the above guidelines and possible solutions, the performance of the embedded system can be improved reaching the target's platform specifications.

Chapter 10

Conclusions and Future work

10.1 Summary of contributions

In this thesis we described an efficient way to improve the performance of the Receptive Fields Cooccurrence Histograms Algorithm on FPGA. We have designed hardware accelerators for specific functions of the algorithm that consume up to 97.8% of its execution time. Our hardware implementations are based on multi-core architectures that can utilize an arbitrary number of processing cores, limited by the device resources. We have demonstrated an overall speedup of upto 15X with the described configuration (16-16-8 cores), but additional performance can be achieved by increasing the number of each modules cores.

Moreover our hardware is not fixed to a specific image size but instead is designed to support images sizes up to High Definition (HD), making it suitable for a variety of applications. Additionally, the number of features and the number of clusters that can be used its also not fixed, which gives as additional flexibility. We have also presented an embedded system prototype for the above algorithm which seems very promising and also gives us a hands-on platform for further experiments.

10.2 Future work

As of this time our system prototype does not meet the desired requirements and in order to do so, additional research is needed. A few things that could be done are the following:

- Regarding Microblaze, a clock rate increase must be achieved in order to have a performance in reasonable margins. We recommend a clock rate of at least 400 MHz. An other choice is to replace Microblaze with a hardcore processor such as PowerPC or ARM Cortex.

- The RFCH Controller could be implemented to use advanced PLB features like Burst Read-/Write and Bus locking, in order increase the I/O performance, since I/O is critical to the specific algorithm.

- For additional speedup regarding the software functions, a floating point arithmetic to fixed point arithmetic conversion could be made.
- The board could be augmented with a camera, in order to capture images and stores them into the compact flash.

Appendix A

Software drivers

In this appendix we list the software drivers that were implemented during the project:

A.1 Low level drivers

The below two functions access the RFCH controller's (PLB Slave) software accessible registers:

```
/* Write to register regNo  */
void vlb_MemWr(int data,int regNo);

/* Read from register regNo */
Xuint32 vlb_MemRd(int regNo);
```

The functions below are used to control the hardware modules:

```
/* Reset the module cs */
void vlb_ChipReset(int cs);

/* Enable the module cs */
void vlb_ChipEnable(int cs);

/* Check the status bit of module cs */
int vlb_ChipStatus(int mode)
```

The functions below implement the block RAM I/O and also handle the data decomposition:

```
void vlb_FeatureArrayToBram(unsigned char **array, int slice_no);

void vlb_BramToFeatureArray(unsigned char **array, int slice_no);

void vlb_ClusterPointArrayToBram(unsigned char **array);

void vlb_BramToClusterPointArray(unsigned char **array);

void vlb_BinnedImageArrayToBram(int *array, int slice_no);

void vlb_BramToBinnedImageArray(int *array, int slice_no);

void vlb_CcountArrayToBram(unsigned long *array);

void vlb_BramToCcountArray(unsigned long *array);

void vlb_CsumArrayToBram(unsigned long **array);

void vlb_BramToCsumArray(unsigned long **array);
```

A.2 High level drivers

The functions below control the hardware modules functionality.

```
void CalculateClusters_hw();

void ClusterFeatures_hw();

void CalculateRFCH_hw();
```

A.3 Image manipulation

```
/* Read an image from the compact flash */
void Load_P3(const char *filename);

/* Convert ASCII image to Binary */
unsigned char* P3toP6_ppm(SYSACE_FILE *f, int *xsize, int *ysize);
```

Bibliography

[1] S.Ekvall, D.Kragic, *"Receptive Field Coocuurrence Histograms for Object Detection"*, in IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 84 - 89, 2005

[2] Altera®, *"FPGA Architecture white paper"*.

[3] Xilinx®, *"DS150 Virtex-6 Family Overview"*.

[4] Xilinx® , *"WP298 Power Consumption at 40 and 45 nm white paper"*.

[5] Sameer A. Nene and Shree K. Nayar and Hiroshi Murase, *"Columbia Object Image Library (COIL-100)"* in Technical Report CUCS-006-96, Department of Computer Science, Columbia University, 1996.

[6] H. Murase and S. K. Nayar, *"Visual learning and recognition of 3-d objects from appearance"*, International Journal of Computer Vision,vol. 14, pp. 5–24, 1995.

[7] A. Selinger and R. C. Nelson, *"A perceptual grouping hierarchy for appearance-based 3d object recognition"*, Computer Vision and Image Understanding, 76(1):83–92, October 1999.

[8] A. Selinger and R. C. Nelson, *"Appearance-based object recognition using multiple views"*, Tech. Rep. 749, Comp. Sci. Dept. University of Rochester, Rochester NY, June 2001.

[9] B. Caputo, *"A new kernel method for appearance-based object recognition: spin glass-Markov random fields"*, PhD thesis, Royal Institute of Technology, Sweden, 2004.

[10] K. Arbter, W. E. Snyder, H. Burkhardt, and G. Hirzinger, *"Application of affine-invariant Fourier descriptors to recognition of 3-d objects"*, IEEE Trans. Pattern Anal. Mach. Intell., vol. 12, no. 7, pp. 640–647, 1990.

[11] R. Singh and N. Papanikolopoulos, *"Planar shape recognition by shape morphing"*, Pattern Recognition, vol. 33, no. 10, pp. 1683–1699, 2000.

[12] Paul Besl and Ramesh Jain, *"Three-dimensional object recognition"*, ACM Computing Surveys, 17(1):75–145, 1985.

[13] M. Swain and D. Ballard, *"Color indexing"*, IJCV7, pp. 11–32, 1991.

[14] B. Schiele and J. L. Crowley, *"Recognition without correspondence using multidimensional receptive fields histograms"*, International Journal of Computer Vision, vol. 36, no. 1, pp. 31–50, 2000.

[15] O. Linde and T. Lindeberg, *"Object recognition using composed receptive field histograms of higher dimensionality"*, 17th International Conference on Pattern Recognition, ICPR'04, 2004.

[16] B. Mel, *"SEEMORE: Combining Color, Shape and Texture Histogramming in a Neurally Inspired Approach to Visual Object Recognition"*, Neural Computation, vol. 9, pp. 777–804, 1997.

[17] P. Chang and J. Krumm, *"Object recognition with color cooccurrence histograms"*, CVPR'99, pp. 498–504, 1999.

[18] D. Lowe, *"Object recognition from local scale-invariant features"*, International Conference on Computer Vision, pp. 1150–1157, 1999.

[19] Changjian Gao and Shih-Lien Lu, *"Novel FPGA based HAAR classifier face detection algorithm acceleration"*, International Conference on Field Programmable Logic and Applications, pp.373 - 378, 2008.

[20] Wenhao He and Kui Yuan, *"An Improved Canny Edge Detector and its Realization on FPGA"*, in Proceedings of the 7th World Congress on Intelligent Control and Automation June 25 - 27, 2008, Chongqing, China.

[21] Christos Gentsos, Calliope-Louisa Sotiropoulou, Spiridon Nikolaidis and Nikolaos Vassiliadis, *"Real-Time Canny Edge Detection Parallel Implementation for FPGAs"*, International Conference on Electronics, Circuits, and Systems (ICECS), pp.499, 2010.

[22] Deepayan Bhowmik, Balasundram P. Amavasai and Timothy J. Mulroy, *"Real-time object classification on FPGA using moment invariants and Kohonen neural networks"*, Proc. IEEE SMC UK-RI 5th Chapt. Conf. Advances in Cybernetic Systems (AICS 2006), pp. 43-48, 2006.

[23] Vinod Nair and Pierre-Olivier Laprise and James J. Clark, *"An FPGA-Based People Detection System"*, in EURASIP Journal on Applied Signal Processing 2005:7, 1–15.

[24] Maissa Ali, Joaquin Sitte and Ulf Witkowski, *"Parallel Early Vision Algorithms for Mobile Robots"*, Proceedings of the 4th International Symposium on Autonomous Minirobots for Research and Edutainment(AMiRE2007), vol. 216, pp.133 -140, Germany, 2007.

[25] J. B. MacQueen, *"Some Methods for classification and Analysis of Multivariate Observations"*, Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability, pp. 1:281–297, University of California Press, 1967.

[26] Richard O.Duda and Peter E.Hart and David G.Stork, *"Pattern Classification 2 edition"*, Wiley-Interscience publications.

[27] Intel® VTune™ Amplifier XE, *"Intel® VTune™ Amplifier XE Documentation"*

[28] CVAP Object Detection Image Database , *"http://www.nada.kth.se/ ekvall/codid.html"*

[29] Xilinx® , *"UG534 ML605 Hardware User Guide"*

[30] Xilinx® , *"DS080 System ACE CompactFlash Solution"*

[31] Xilinx® , *"LibXil FATFile System(FATfs)"*

[32] Xilinx® , *"UG463 OS and Libraries Document Collection"*

[33] Micron® , *"512MB (x64, SR) 204-Pin DDR3 SDRAM SODIMM Features"*

[34] Xilinx® , *"UG081 Microblaze processor reference guide"*

[35] Netpbm library , *"Netpbm user manual"*, in http://netpbm.sourceforge.net/doc/

[36] Xilinx® , *"DS561 PLB46 Slave Single"*

[37] Xilinx® , *"UG448 A Guide to Profiling in EDK"*

[38] LibFixmath , *"http://code.google.com/p/libfixmath/"*

[39] Xilinx®, *"Zynq-7000 EPP Product Brief"*

[40] Xilinx®, *"DS565 PLBV46 Master Burst"*

[41] Xilinx®, *"DS769 LogiCORE IP AXI Slave Burst "*