Technical University of Crete

Electronic and Computer Engineering Department

Microprocessor & Hardware Laboratory

High Performance Low Power Embedded Vision Systems

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRONIC AND COMPUTER ENGINEERING

OF TECHNICAL UNIVERSITY OF CRETE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Antonis S. Nikitakis

Chania 2013

Abstract

Human vision is a complex combination of physical, psychological and neurological processes that allow us to interact with our environment. We use vision effortlessly to detect, identify and track objects, to navigate and to create a conceptual map of our surroundings. The goal of computer vision is to design computer systems that are capable of performing these tasks both accurately and efficiently at real-time and using limited resources. There are numerous computer vision systems today in various application fields that demand very fast and accurate detection of certain points in video or live streams.

Considering the multimedia and the upcoming wearable computing field, one very important challenge is the implementation of fast and detailed Object Detection/Tracking and object Recognition systems. In particular, in those systems, it is highly desirable to detect and locate certain objects within a video frame in real time but also using minimum energy. Although a significant number of Object Detection, Tracking and Recognition schemes have been developed and implemented, triggering very accurate results, the vast majority of them cannot be applied in state-of-the-art mobile multimedia devices; this is mainly due to the fact that they are highly complex schemes that require a significant amount of processing power, while they are also time consuming and very power hungry.

In this thesis we present three different approaches in building high performance Embedded Vision Systems, while our focus is both real time performance as well as low power consumption. Following a certain set of principles identified after the analysis of numerous vision system we have implemented three state of the art computer vision schemes utilizing state-of-the-art FPGA devices along with embedded processors: The OpenSurf which is a feature extraction algorithm, the RFCH which is a state of the art object detection algorithm, and the OpenTLD which is a self-trained stable tracking algorithm.

We clearly demonstrate that such complex vision tasks can, probably for the first time, be addressed by a single chip solution running on minimal power; this is

achieved by exploiting the availability of custom hardware coupled with a low power embedded CPU. In this thesis we present such a single chip prototype powered by the recreantly announced Xilinx Zynq-7000 single chip device featuring a dual-core ARM CPU together with reconfigurable logic in the same silicon.

Acknowledgements

I would like to thank Dimitrios Bouris, Savvas Papaioannou and Theofilos Paganos for the excellent collaboration and their valuable contribution in this thesis work. I would also like to thank my advisor Dr. Ioannis Papaefstathiou for his advice and guidance during all my years in my Ph.D.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Computer Vision

Computer vision (or machine vision) is the science and technology that enables machines to solve particular tasks or make decisions by extracting information from an image. Computer Vision is relatively new, although rapidly developing, field of study with strong scientific and industrial support. As an applied science field, computer vision seeks to apply its theories and models to the construction of real-wold 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 certain other fields such as:

- Artificial intelligence

  is a branch of computer science and technology that studies and develops intelligent machines and software which also applied in computer vision tasks.

- Image processing and image analysis

  focuses on 2D images, how to transform one image to another, (e.g. by pixel-wise operations such as contrast enhancement, local operations such as edge extraction or noise removal, or geometrical transformations such as rotating the image). This characterization implies that image processing/analysis neither require assumptions nor produce interpretations about the image content.

- Pattern recognition

  extracts information from signals mainly based on statistical approaches. A significant part of this field is devoted to applying these methods to image data. Pattern recognition algorithms generally aim to provide a reasonable answer for all possible inputs and to perform "most likely" matching of the inputs, taking into account their statistical variation.

- Neural Networks

  refer to artificial neural networks, which are composed of artificial neurons or nodes. These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained using a dataset.

1.2. Motivation

There are numerous FPGA systems utilizing computer vision algorithms that have been presented over the last years. The vast majority of them are custom-fitted hardware implementations requiring extensive development time and deep expertise in FPGA design. They are usually not customizable and they cannot be connected to existing systems (purely hardware or even CPU-based ones) while they cannot be utilized in conjunction with the standard vision software libraries (like openCV). By analyzing the literature we also see that the current FPGA-based approaches are usually targeted to older and well studied vision methods rather than blending in with today's trends in computer vision; one important reason for this is the lack of a unified platform that allows for the an efficient hardware/software co-design approach tailored to the needs of the computer vision applications.

In this thesis we are motivated from this lack of a unified architecture which can bind together the software and the hardware world in computer vision, without any compromises.

From the software side, this architecture would be able to support any modern computer vision algorithm and library like OpenCV while providing a solid and

efficient way of communication with hardware accelerators. On the hardware side it should offer support to different FPGA devices and a standardized way of communication with the software in order to offer the flexibility of customization and long term compatibility with well established software libraries.

The architecture will be the basis for implementing numerous high-end novel vision embedded systems which will be faster and more power efficient than the existing solutions.

1.3. Contribution

The contributions of this work come mainly from the implementation of three computer vision systems which are all faster while consuming less energy than any existing approaches. This has been achieved by applying certain principles breifly mentioned at the end of this section and analytically described in Chapter 8 that were proposed for the first time and which can be utilized in the implementation of numerous similar systems.

The first system implemented is based on the SURF (Speeded-Up Robust Features) detector introduced by Bay, Ess, Tuytelaars and Van Gool; [BAY et al. 2006] this algorithm is considered to be the most efficient feature detector algorithm available.

- This was at the time of publication [MPOURIS et al. 2010], to the best of our knowledge, the first implementation of this scheme in an FPGA.
- Our innovative system can support processing of standard video (640 x 480 pixels) at up to 56 frames per second while
- Our system outperforms a state-of-the-art dual-core Intel CPU by at least 8 times. Moreover, the proposed system, which is clocked at 200MHz supports constantly a frame rate only 20% lower than the peak rate of a high-end GPU executing the same basic algorithm which is powered by 128 floating point CPUs, clocked at 1.35GHz
- The GPU consumes more than 200W, the dual core Intel 40W (while being 8 times slower) while our systems consumes less than 20W; thus it is in general

about one order of magnitude more energy efficient than both a CPU and a GPU.

- Unlike all existing systems, our system keeps a constant performance even though the detected points are increased.

Furthermore we implement in reconfigurable hardware one of the most efficient Object Recognition Algorithms, the Receptive Fields Cooccurrence Histograms (RFCH) one. This is, to the best of our knowledge, the first system presented (at the time of the publication [NIKITAKIS et al. 2012]) that can execute the complete complex object recognition task at a multi frame per second rate while consuming minimal amounts of energy, making it an ideal candidate for future embedded multimedia and wearable computing systems.

- Our low-power embedded reconfigurable system is at least 15 times faster than the software implementation on a low-voltage high-end CPU, while consuming at least 60 times less energy.
- Our novel system is also 88 times more energy efficient than the recently introduced low-power multi-core Intel devices which are optimized for embedded systems.
- Our system it is not fixed to a specific image size but instead it is designed to support image sizes up to High Definition (HD), allowing it to be utilized in numerous distinct multimedia applications.
- Additionally, the number of features and the number of clusters that are supported are not fixed either and they can be altered by just changing the software part of the system which offers even greater flexibility

Considering the OpenTLD algorithm, we present a novel embedded system which is implemented in an FPGA device and also utilizes a low power on-chip ARM processor. In our proposing architecture we accelerate the bottleneck of the algorithm by designing and implementing a high bandwidth distributed memory sub-system which is independent of the various software parameters.

Based on our real-world measurements our embedded system is more than 23 times faster than a state-of-the art Intel CPU and marginally faster than a highly parallel Graphical Processing Unit (GPU) while:

- It is at least 40x more energy efficient than both the Intel CPU and the GPU.
- it is portable to various FPGA devices due to its transparent bus utilization
- It is highly scalable and can be seamlessly adapted for different computer vision schemes that are utilizing the Random Forest structures.

Finally we propose a unified reconfigurable hardware architecture which is able to support different computer vision algorithms while also being software-customizable. In such a platform the user is able to reprogram the FPGA device by altering the bitstreams of various accelerators depending on the application needs while leaving the software untouched. We utilize a “middleware” driver for this purpose, which is able to transparently accelerate on-demand (by a regular function call) the hot spots of the underlying algorithm. As a proof of concept we implemented both RFCH and OpenTLD in a very low cost development board consisting of a dual-core CPU and a reconfigurable fabric on the same die.

Chapter 2

Feature Detection Object Recognition and Object tracking

In this chapter we are describing the basic challenges in the computer vision domain and define the problems that the three proposing hardware architectures of this thesis are addressing. The first one is the Feature detection problem, the second one the object detection/recognition problem and the third one the object tracking problem. In the end of the chapter we present some of the most important FPGA-accelerated object detection/recognition systems in the literature.

2.1. Feature Detection

In computer vision and image processing the concept of feature detection refers to methods that aim at computing abstractions of image information. Those abstractions lead to a significant reduction of the initial image information and enable local decisions at every image point, whether there is an image feature of a given type at that point or not. The resulting features will be subsets of the image domain forming a more compact representation of it;they are often in the form of isolated points, continuous curves or connected regions. [WEB_FEATURE]

There is no universal or exact definition of what defines a feature, and the exact definition often depends on the problem or the type of application. Given that, a feature is defined as an "interesting" part of an image, and features are used as a starting point/input for many computer vision algorithms. Since features are used as the starting point and main primitives for subsequent algorithms, the overall algorithm will often only be as good as its feature detector. Consequently, the desirable property for a feature detector is repeatability: whether or not the same feature will be detected in two or more different images of the same scene. [WEB_FEATURE]

Feature detection is a low-level image processing operation. That is, it is usually performed as the first operation on an image, and examines every pixel to see if

there is a feature present at that pixel. If this is part of a larger algorithm, then the algorithm will typically only examine the image in the region/domain of the features.

2.1.1. Types of image features

Depending on the algorithm and the aim of the target application we have different types of features. Each type may be more appropriate than other types for a specific application.

- Edges
    - Edges are points where there is a boundary (or an edge) between two image regions (i.e the color intensity rapidly changes). In general, an edge can be of almost arbitrary shape, while they are usually defined as sets of points in the image which have a strong gradient magnitude.
- Corners / interest points
    - The terms corners and interest points are used somewhat interchangeably and refer to point-like features in an image, which have a local two dimensional structure. The name "Corner" arose since early algorithms first performed edge detection, and then analyzed the edges to find rapid changes in direction (corners). [WEB_FEATURE]
- Blobs / regions of interest or interest points
    - Blobs provide an additional description of image structures in terms of regions, as opposed to corners that are more point-like or edges that are line-shaped. Blob detectors are either based on differential methods (i.e derivatives of the function with respect to their position) or based on local extrema (i.e finding the local maxima and minima of the function). Nevertheless, blob descriptors often contain a preferred point (similar to interest point). This could be a local maximum of an operator response or a center of gravity which means that many blob detectors may also be regarded as interest point operators. [WEB_FEATURE]

The most common feature detectors used today are: Harris interest point detector [HARRIS et al. 1998], Scale Invariant Feature Transform (SIFT) [LOWE et al. 1999] and Speed-Up Robust Features (SURF) [BAY et al. 2006]. While SIFT and SURF are invariant to illumination, rotation and scale, Harris interest point's detector is not invariant to scale. On the other hand, Harris detector is faster than both SIFT and SURF, however it is less accurate.

2.2. Object recognition

Object recognition is the task of finding a given object in an image or video sequence. Humans recognize a multitude of objects in images with little effort, despite the fact that the image of the objects may vary somewhat in different viewpoints, in many different sizes / scale or even when they are translated or rotated. Objects can even be recognized when they are partially obstructed from view. This task is still an important challenge for computer vision systems in general. [WEB_RECOGNITION]

Recognizing objects is a central challenge in the field of computer vision and autonomous robotics. Although a significant amount of work has been reported, the proposed methods still differ significantly between these two research areas [EVKALL et al. 2005]. Let's consider an autonomous robot scenario. Here, the robots are about to operate in a complex indoor environment (office or home), which means that it is very difficult to model all possible objects that the robot is supposed to manipulate. Furthermore, the applied object recognition algorithm has to be robust to outliers and changes commonly occurring in such a dynamic environment, such as illumination, scale and rotation changes.

2.3. Object Tracking

The problem of long term visual tracking is very important in numerous application domains including surveillance, security, augmented reality and multimedia. For example the very recent introduction of Google Glass [WEB_GLASS] initiates a new era of wearable computing; small, yet powerful embedded devices will have to

perform demanding computer vision tasks like object detection and tracking in a continuous process while displaying the results in a novel way. Such embedded environments demand moving objects like persons, traffic signs etc to be located and/or tracked in real time while consuming minimum amount of power.

Moreover object tracking is very popular among multimedia applications like video stabilization which reduces the shaking that is associated with the motion of a camera during exposure. The "match moving" cinematic technique is another multimedia application that allows the insertion of computer graphics or video footage into live-action footage in the correct position, scale, orientation, and motion relative to the photographed objects in the shot. [WEB_MATCH].

The tracking process can be very difficult when the objects are moving fast relative to the frame rate and the object has to be tracked for a long period. "Long-term" object tracking refers to such circumstances where there are large video sequences that contain frame-cuts and fast camera movements and thus the object may temporarily disappear from the scene. The two most critical points in all those applications are the accuracy of the system as well as its real time performance at high resolutions.

All those tracking issues are efficiently addressed by the very promising and newly introduced OpenTLD algorithm [KALAL et al. 2009]; OpenTLD is based on the Random Forest classifier introduced by Breiman Leo [BREIMAN 2001]. Even though the utilization of a Random Forest classification approach is being widely used in computer vision, it has a certain disadvantage over similar classification schemes; it cannot be efficiently parallelized at a fine grain (i.e. at the Random Forest access level).

2.4. FPGA accelerated object detection/recognition systems

There is significant amount of research work that has been done over the last years considering the hardware implementation of object detection algorithms. In this subsection we give an overview of some of the most important research results and/or systems in the wide area of object detection/recognition devices as we

cannot directly relate them to our work. The closely related work from the literature is presented in Chapter 3.

The proposed schemes in the literature usually utilize FPGAs and try to exploit the inherent parallelism of those algorithms. The achieved speedup and the low power consumption over the sequential version of the algorithm, usually comes with a cost: increased development time and usually complicated FPGA design which requires expertise in custom hardware design.

The existing systems are based on well established pattern recognition algorithms for the classification stage, such as neural networks, Support Vector Machines, and the very popular Viola-Jones detection framework. They mostly utilize a sliding window approach to search for objects of various sizes, while they usually utilize 320x240 or 640x480 images or video streams.

The detection framework by Viola and Jones [VIOLA and JONES, 2001] is maybe the most popular approach found in the literature, used for 2D object detection, and thus it is also adopted in many hardware accelerated schemes. The Viola and Jones framework is proved to be hardware friendly in resource-limited devices such as FPGAs, as it requires additions and a few multiplications while its internal accuracy demands can be usually addressed by a fixed-point arithmetic scheme.

An early attempt in hardware accelerated object detection was the work of [MCCREADY, 2000]. The author proposed a custom face detection algorithm based on a neural network-like classifier which is trained from a large number of positive face and negative (non-face) examples. The system is implemented on 9 boards of an TM-2a system which is a configurable multi-board FPGA platform and uses 31500 Altera Logic Cells. It handles 320x240 images and achieves a frame rate of 30 fps while having detection accuracy comparable to other existing methods at the time of publication.

The work from [CHO et al, 2009] demonstrates a parallel implementation of the Viola and Jones algorithm on FPGA devices, achieving real-time performance of 23 fps for 320x240 images. In this work the authors present a custom hardware

architecture for a face detection system which is based on the AdaBoost machine learning algorithm which uses the Haar features. The proposed architecture is implemented in a Xilinx Virtex-5 FPGA and achieves up to 35 times increase of system performance over the equivalent software implementation. They achieve up to 7.51 fps frame rate for 640×480 images and up to 28.8 fps for 320×240 images. The presented system is a well-fitted FPGA implementation but it lacks the expandability and the flexibility of a programmable embedded system.

The authors from [BROUSSEAU and ROSE, 2012] propose a hardware architecture for object detection, that is fully compatible with the object classifiers used in the OpenCV computer vision library. Their hardware scheme is designed in such a way so as to accept the same inputs, and produce the same output with the OpenCV implementation of the Haar Feature-based Cascade Classifier for Object Detection. Thus it could be used by developers who are using the OpenCV library in order to achieve better performance and low power consumption in a transparent way. Their hardware implementation outperforms a modern mobile processor by a factor of 59 times, and it is 13.5 times more energy-efficient. However their system is tested for relatively low resolution images of 320x240 pixels.

In [LAIKA et al. 2010] the authors present a real-time object detection system targeted for moving robot applications. They utilize an optical flow-based method for the detection process and their system is powered by a Virtex-5 FPGA from Xilinx with an embedded PowerPC Processor. The proposed algorithm estimates and compensates robot's ego-motion in order to efficiently detect and localize objects while it is moving. They use two alternative object detection methods: a 2D-histogram-based clustering and a motion compensation method based on frame differencing and they achieve detection rates up to 81%. They perform the processing at a real time rate of 31 fps at a resolution of 640x480 pixels.

The work from [DUC et al. 2011] proposes an FPGA-based low power object detection system tailored for Wireless Multimedia Sensor Networks. The system is comprised of two blocks, the network processor block executing the basic operations and the FPGA block detecting and extracting the objects of interest in real-time. In

that way their system transmits only the updated object and not the raw image. They prototyped their scheme by using a Xilinx Spartan-3 FPGA device. They claim that their proposed architecture helps to reduce the total energy consumption by approximately twenty times in comparison with the energy consumed for a raw image transmission.

In [HIROMOTO et al, 2009] the authors propose a hardware architecture for object detection based on the AdaBoost learning algorithm with Haar-like features. The hot spots of the algorithm are implemented in parallel hardware modules exploiting the cascade structure of the classifier. They assign more resources to the earlier and most frequently used classifiers in the cascade in order to achieve a minimum design footprint without compromising the performance. In addition, they implement the proposed architecture on a Virtex-5 FPGA to show that it achieves real-time object detection at 30 fps on VGA video streams.

In [KYRKOU et al., 2013] the authors propose an FPGA hardware architecture for object detection that utilizes a search reduction approach in order to speedup a Support Vector Machine classification-based scenario. The authors exploit the depth and edge information when their system is integrated into an existing 3D vision system in order to build an efficient embedded object detection system. Their system was evaluated using images of various sizes, with results indicating that the proposed architecture is capable of achieving real-time frame-rates for a variety of image sizes (271 fps for 320x240, 42 fps for 640x480, and 23 fps for 800x600) while reducing the false positive rate by 52% in a face detection application when compared with existing solutions.

In [CHE et al., 2010] the authors propose the implementation of a face detection algorithm on FPGA using a software/hardware co-design approach. Their system is implemented on an Altera Cyclone II FPGA which instantiates a Nios embedded soft-core processor. The processor handles the lightweight parts of the algorithm as well as the coordination of the whole system. The more demanding parts of the algorithm are implemented in the FPGA device. A dedicated hardware accelerator was used for the processing of the skin color operation and a distributed look-up

table method was designed to accelerate the floating-point multiplication in a color space transformation unit. The proposed system is clocked at 100 MHz and handles each 640 x 480 image in 96 ms.

Chapter 3

Related work

In this chapter we present the related work considering our three innovative embedded systems analytically described in Chapters 4,5 and 6. In Section 3.1 we present the hardware schemes which are similar to the openSURF system, implementing feature detectors (such as SIFT). In section 3.2 we give an overview of the most important object/detection and recognition hardware systems. Finally in Section 3.3 we give an overview of systems utilizing object tracking schemes while also involving the Random Forest classifier.

3.1. OpenSURF

There are numerous FPGA-based feature detectors that have been presented in the past; the majority of them implement Tomasi and Kanade's corner detector [TOMASI, 1991]. Within this scheme, the best tracking method turns out to be the one proposed by Lucas and Kanade in 1981 [LUCAS and KANADE, 1981]; the Lukas and Kanade method defines the measure of match between certain fixed-size feature windows in the past and the current frame as the sum of squared intensity differences over the windows. It also examines far fewer potential matches between the images than any other technique and it can be generalized to handle rotation, scaling and shearing.

The above method is then used in a series of other systems for efficient object tracking. The work form [BENEDETTI et al. 1998], proposes a system for real-time detection of 2-D features on an FPGA which is able to track image features in real-time applications like autonomous vehicle navigation. Their method optimizes Tomasi and Kanade's method and it is suitable for implementation on a low-cost reconfigurable device.

In [FIORE et al. 1998], the authors use several heuristics on top of in the Kanade-Lucas-Tomasi method. They propose an efficient, high-performance FPGA design that does not sacrifice the detection performance. Their high level design couples

the FPGA with a conventional digital signal processor to enable real time processing in video streams.

In [BISSACCO et al. 2006] the authors implement a fast visual feature tracking system which takes advantage of a dedicated Xilinx VIRTEX FPGA device to perform the computationally intensive step of selection. Then, a software system uses the output of the hardware module to complete the visual tracking algorithm.

Moreover, S. Se, in their ExoMars system [SE et al. 2005] utilizes an FPGA which implement the scale invariant SIFT algorithm form [LOWE et al. 1999] for localization and terrain modeling. Their system can compute SIFT locations and descriptors for a 640x480 image in 60 ms (i.e. at 16 fps) using a Xilinx Virtex II FPGA.

A similar system has been implemented by Bouganis et.al. [BOUGANIS et al. 2004]; they compute corner locations using complex steerable wavelets at four different scales and orientations and they can achieve up to 25 fps when implemented on a state-of-the-art FPGA. As the standard version of SURF is considered to be significantly more robust against different image transformations than SIFT, we believe it will not be fair to compare our SURF implementation with the implementations of SIFT-based schemes; in any case our system outperforms all the known feature-based detectors implemented in FPGAs in terms of performance since it is the only one supporting 56 frames per second.

To the best of our knowledge, this is the first implementation of SURF in hardware (at the time of publication: [BOURIS et al. 2010]); the only other non-standard software implementation known is the one presented in [TIMOTHY et al. 2009] in which they mainly list the performance of an adapted version of the algorithm when ported to a highly parallel high-end GPU.

3.2. RFCH scheme

Even though no complete embedded, low-power and multi-frame per second object recognition system exists, there are several FPGA-based systems implementing certain face recognition algorithms as well as some hardware systems executing

specific sub-parts of the object recognition algorithms that are related, in a certain manner, to our work.

In [CHANGJIAN et al. 2008] the authors present a novel approach, utilizing a state-of-the-art FPGA, so as to accelerate the Haar-classifier face detection algorithm. By utilizing a large number of parallel arithmetic units in the FPGA they achieved real-time performance, with very high detection rates and very low false positives. Their implementation is tailored to a HiTech Global PCIe card that contains a Xilinx XC5VLX110T FPGA device. Moreover in [KYRKOU et al. 2010] another Haar-based face detection scheme is described which outperforms all the existing such schemes implemented in FPGAs. However, all those systems are optimized for face-detection and cannot be efficiently applied to general object-recognition.

The authors in [WENHAO et al. 2008] proposed a novel self-adaptive Canny edge detection scheme while they also present an FPGA implementation optimized for mobile robotic systems. Their system utilizes an Altera Cyclone EP1C60240C8 and can detect the edges of a certain, pre-defined, object on a grey-scale image at an analysis of 360x280 in 2.5ms (or in other words at a speed of 400 frames per second). In [GENTSOS et al. 2010] the authors present another implementation of the Canny edge detector that processes 4-pixels in parallel; this approach increases the throughput of the design without increasing the required on-chip cache memories. By increasing the parallelism of their scheme their can processes high resolution images (up to 1.2Mpixels) in 3.09ms (i.e. at about 300 frames per second) when their scheme is implemented on a Xilinx Spartan-6 FPGA clocked at 200MHz. However, their system implements only the edge detection task while the rest of the object recognition process is not supported or even discussed.

In [DEEPAYAN et al. 2006] a hardware implementation of an object classification system based on moment invariants and Kohonen neural networks is presented capable to classify objects in real-time. The authors implemented the classification phase in hardware while leaving the training of the Kohonen network into software; in particular the computation of the 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; unfortunately no specific performance numbers are given.

In [VINOD et al. 2005] the authors present an FPGA-Based People Detection System. They use JPEG-compressed frames from a network camera which after pre-processing (i.e. feature extraction), are sent to a machine-learning detector, implemented on a Virtex-II 2V1000; the FPGA executes the actual 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 when clocked at 75 MHz.

In [GOSHORN et al. 2010] the authors present an object detection system that can detect a single object at a rate of 266 frames per second. However, they did not present any data about its accuracy and since they use a very poor correlation method based on the sum of absolute differences (SAD), the accuracy of their system is heavily questioned; moreover, their device can detect only a pre-defined single object in a single scene while they only roughly localize it (i.e. localize only the center of the object and they do not report any bounding box).

Finally in [SHOTTON et al. 2011] the authors propose a new method to predict 3D positions of body joints from a single depth image at up to 200fps on consumer hardware. However they use a depth camera such as Microsoft Kinect [XBOX] which consists of an infrared laser projector combined with a monochrome CMOS sensor. They also don't generalize their method to other object detection tasks that may be useful in multimedia systems.

When compared with all those existing systems our approach has certain significant advantages such as:

1) It is the only one supporting the complete general, multi-object recognition and localization task at more than one frame per second.
2) This is, to the best of our knowledge, the only embedded system that has been specifically designed so as, not only to be real-time, but also to

consume as less energy as possible, in order to address the needs of today's embedded multimedia devices.

3) It is the only system that can work simultaneously on multiple features (i.e. 7 features) which significantly increase the robustness of the system while still supporting a multi frame per second rate in real-world environments.

4) Even when compared with the different face detection systems, it is the only one performing efficiently in hardware the on-line training phase utilizing only a single training sample per object; the Haar-based systems need hundreds of training samples per object and thus they do the training off-line which severally limits their efficiency.

5) The algorithm utilized is probably one of the most accurate generalized object recognition algorithms presented so far as described in [EKVALL et al. 2005].

Based on the above, we believe that this is the first system addressing all the needs of the real-time embedded multimedia devices, recently introduced, that involve complex object recognition tasks.

3.3. OpenTLD scheme, related work

3.3.1. The OpenTLD/Random Forest related work (hardware)

[OSMAN 2009] presents the implementation of an object recognition system based on the Random Forest Classifier and targeted for an FPGA platform. The specified approach utilizes a Logarithmic Number System while their architecture executes the algorithm in an optimized, yet sequential manner. The author assumes that everything is on on-chip memories and can be accessed in a single cycle whereas he does not present any performance (i.e. latency or bandwidth) results nor he tries to parallelize the classifier in any manner.

In [BECKER et al. 2011] the authors present a hardware architecture that implements the most compute-intensive part of the OpenTLD object tracking algorithm. They exploit another approach for parallelization which is inherent in some random forest classifiers: they access different forests simultaneously. They utilize one classifier with ten decision trees that they are all processing the same input. They actually use multiple copies of the same image data (within a sub-window) for each decision tree. They get a 5x speedup when comparing their hardware performance with that of their own custom software implementation. Their initial approach is indeed interesting and in order to get even higher performance they are implementing, on the same device, more than one classifiers each running on a different input. The authors assume that they can easily supply data to the 20 different classifiers that can fit on an FPGA from a single on-chip integral memory (no further details about it are given); however, since the problem is memory bound further studies are needed in order to investigate whether the specified memory bandwidth can indeed be supplied by a single on-chip memory module.

In comparison with all those systems, our approach is the only one which is totally application-independent whereas, to the best of our knowledge, this is the first embedded scheme that allows for the efficient parallelization of the Random Forest classification problem, while being significantly faster and more energy efficient that the existing hardware approaches.

3.3.2. Random Forests implementation in software

In this subsection we present an overview of some existing object detection schemes based on the Random Forest classifier implemented in software, in order to support that the specific approach is giving very promising results. Thus it is worth exploring the possibility of developing an application independent hardware accelerator for the random classifier, like the openTLD presented in this thesis.

The authors of [MAREE et al. 2004] propose a certain scheme for image classification based on this algorithm. Their method operates directly on pixel values and does not require feature extraction. It combines a simple local sub-window extraction technique with the induction of ensembles of randomized decision trees. The same authors in [MAREE et al. 2005], as well as the authors in [SUMALATHA et al. 2011], improve their previous work by using a novel technique of extracting sub-windows that takes into account certain transformations which are independent of the input parameters. During training, certain sub-windows are randomly extracted from the training images and a model is constructed utilizing certain machine learning methods which are applied to the transformed versions of these images.

The authors of [KALAL et al. 2009] investigate the problem of robust, long-term visual tracking of unknown objects in unconstrained environments. They propose a new approach, called Tracking-Modeling-Detection (TMD) that closely integrates adaptive tracking with online learning of the object-specific detector. In their object detector implementation, they use a sequential randomized forest classifier. The forest consists of several trees, where each of them is built from a certain group of features. Every feature in the group represents a measurement taken at a certain level of the tree. The extracted features are randomly partitioned into several same-size groups. Each group represents a different view of the patch appearance. The response of each group is represented by a discrete vector that is called a “branch”.

When trying to efficiently implement, in and embedded system, all the above schemes, although there is some inherent parallelism, it is very hard to perform a memory partitioning and assignment which will allow for the effective exploitation

of this parallelism. In this thesis we present a scheme that addresses this memory problem and it is targeted for reconfigurable/embedded devices. Our innovative approach results in an efficient parallelization of the Random Forest classification method, while keeping the flexibility to support different similar algorithms.

Chapter 4

OpenSurf System

SURF (Speeded Up Robust Features) is a robust local feature detector, first presented by Herbert Bay et al. in 2006. OpenSURF [WEB_SURF] is a dedicated library implementing the SURF algorithm as an open source project. SURF can be used in computer vision tasks like object recognition. In this chapter we present an overview of the algorithm itself while also demonstrating our hardware architecture triggering very high speeds combined with low energy consumption while targeted to FPGA devices.

4.1. The SURF algorithm

The SURF detector algorithm tries to localize the "interest points". An interest point is a point in the image which: (a) has a clear, preferably mathematically well-founded, definition, (b) has a well-defined *position* in the image space, (c) the local image structure around it is rich in terms of local *information contents*, so the use of interest points simplifies further processing in the vision system, (d) is *stable* under local and global perturbations in the image domain, including deformations as those arising from perspective transformations such as scale changes, rotations and/or translations as well as illumination/brightness variations. Moreover, the notion of the interest point includes an attribute of *scale*, so as to allow it to compute interest points from real-life images as well as under scale changes.

The SURF algorithm consists of the following distinct processing steps:

1. Calculate the integral image representation.
2. Calculate the determinant response map into a certain scale-space level.
3. Perform scale normalization of the determinant response map
4. Perform non maximal suppression in order to localize the interest points in the scale-space.
5. Compare the non-maximal suppression result with a predetermined threshold.

6. Perform Interest point localization into the scale-space based on a certain equation
7. Calculate the orientation of each interest point.

The high performance and accuracy of SURF can mainly be attributed to the use of an intermediate image representation known as 'Integral Image' [VIOLA and JONES 2003]. Moreover, the efficiency of the SURF detector is also due to the use of the Hessian-matrix approximation which is claimed to provide highly accurate results. The Hessian matrix of a continuous function f is the matrix of the partial derivatives of function f as described in formula (1), whereas the determinant of the Hessian matrix is calculated based on equation (2).

$$H(f(x,y)) = \begin{bmatrix} \frac{\theta^2 f}{\theta x^2} & \frac{\theta^2 f}{\theta \chi \theta y} \\ \frac{\theta^2 f}{\theta \chi \theta y} & \frac{\theta^2 f}{\theta y^2} \end{bmatrix} \quad (1)$$

$$\det(H) = \frac{\theta^2 f}{\theta x^2}\frac{\theta^2 f}{\theta y^2} - \left(\frac{\theta^2 f}{\theta \chi \theta y}\right)^2 \quad (2)$$

Additionally, the necessity of recognizing objects under different scales leads to the introduction of the scale-space concept [WEB_SCALES]. A scale-space is a continuous function which is used so as to find the maximum and minimum values across all possible scales [WITKIN 1983]. The SURF scale-space is created by applying kernels (i.e. box filters) of increasing size to the original image. This approach further increases the performance since: (a) it allows for multiple layers of the scale-space pyramid to be processed simultaneously (b) it avoids the need to subsample the image.

The scale-space is divided into a number of octaves, where an octave refers to a series of response maps computed by specific filters. The construction of the scale space starts with a 9x9 filter, which calculates the determinant response of the image for the smallest scale (this filter corresponds to a real valued Gaussian distribution with σ=1.2 where σ is the "scale" of the filter). Each octave is divided into four intervals and the filter size is given by equation (3). Subsequent layers are obtained by up-scaling the filters while maintaining the same filter-layout ratio. As

the filter size increases, the value of the associated Gaussian scale is also increased and it is calculated by formula (4):

$$\text{Filter Size} = 3(2^{\text{octave}} \text{xinterval} + 1) \quad (3)$$

$$\sigma_{\text{approx}} = \text{Current Filter Size} \cdot \frac{\text{Base Filter Scale}}{\text{Base Filter Size}} =$$

$$= \text{Current Filter Size} \frac{1.2}{9} \quad (4)$$

The determinants are computed for certain pixels of the image. The structure and the size of the different filters[1] determine which rows and columns (initial and final) have to be processed. This process continues in certain steps which are called sampling intervals.

After the four entries of the Hessian matrix are calculated, for each scale-space location, they are scale-normalized. This scale-normalization of the determinant response map is achieved by dividing these entries by the filter area.

The final part of the detection consists of the interest point localization task. This task is divided into three sub-steps. Firstly, the determinants are compared to a predetermined threshold and the actual localization is not performed for values under this threshold. When this threshold is increased the detected interest points are decreased, while a lower threshold allows for more interest points to be reported. Then, the non-maximal suppression task is performed in order to find a more refined set of candidate interest points. In order to do so each pixel in the scale-space is compared to its 26 neighbors, comprised of the 8 points in the native scale and the 9 points in each of the scales above and below it (in other words the algorithm performs a non-maximal suppression in a 3x3x3 neighborhood). Figure 4.1a describes then non-maximal suppression step. Essentially, the algorithm finds

[1] there is one filter for each octave

the maximum value and if it is less than the threshold, the localization is not performed.

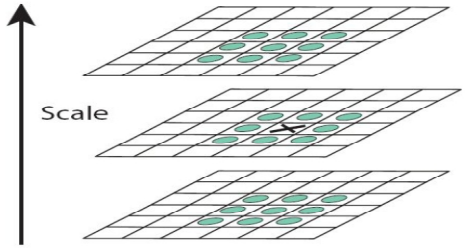

Figure 4.1a. Non-Maximal Suppression

The final sub-step of the interest point localization task involves the interpolation of the nearby data in order to find location of the interest points in both the actual space and the scaled one with sub-pixel accuracy. This is done by utilizing a 3D quadratic scheme as proposed by Brown [BROWN and LOWE 2002]; as a result we express the determinant of the Hessian function H(x,y,σ), using Taylor expansion based on the following formula:

$$H(x) = H + \frac{\theta H^T}{\theta \chi} x + \frac{1}{2} x^T \frac{\theta^2 H}{\theta \chi^2} x \quad (5)$$

The interpolated location of the extremum (i.e. the candidate interest point reported by the non-maximal suppression task), x=(x,y,σ), is calculated by taking initially the derivative of this function and setting it to zero such that:

$$x = - \frac{\theta^2 H^{-1}}{\theta \chi^2} \frac{\theta H}{\theta \chi} \quad (6)$$

The derivatives are approximated by the finite differences of the candidate interest point's neighboring determinant values. The result of equation (6) is a vector with three values. These three values determine the exact position of the interest point (i.e. x,y,scale). In case all three values are less than 0.5 they are adjusted as following: x and y are multiplied with the sampling interval, the result of the multiplication is added to x and y respectively and the final results are rounded to the nearest integers. The new scale is computed based on equation (4); in this

formula the requested filter size is computed by equation (3) while the interval value is the actual scale of the reported point.

The SURF descriptor reports how the pixel's intensities are distributed within a scale-dependent neighborhood consisting of each interest point detected. The extraction of the descriptor is divided into two distinct sub-tasks. Firstly, each interest point is assigned an orientation and based on that a scale dependent window is constructed. From this scaled window the algorithm extracts a 64-dimensional vector. All the descriptor's calculations are based on certain measurements which are relevant to the detected scale so as to achieve scale-invariant results.

In order to achieve the requested invariance to image rotation each detected interest point is also assigned a reproducible orientation; the extraction of all the descriptor components is performed with regards to this orientation/direction so it is important that it is repeatable under varying conditions. In order to determine the orientation, the concept of Haar Wavelets is used.

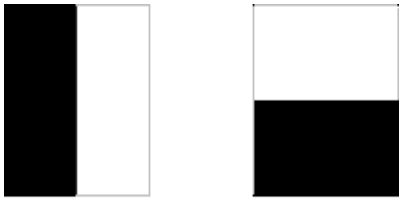

Figure 4.1b. Haar Wavelets

Haar wavelets are filters which are used in combination with integral images in order to increase the robustness of the results and decrease the computation time for reporting them. These simple filters, demonstrated in Figure 4.1b, are utilized in order to find the gradients' values in the x and y directions. The left filter computes the response in the x-direction and the right in the y-direction. Weights are 1 for black regions and -1 for white.

All the Haar wavelet responses of size 4σ are calculated for a set of pixels that are located within a radius of 6σ from the interest point; σ refers to the scale of the detected interest point (i.e. σ is the rounded value of the decimal value of the scale used). The computed responses for each of these pixels form the orientation angle

of a vector. Then a simple classification operation is applied to all those angles (using a sliding window as the one demonstrated in Figure 4.1c), so as to create the dominant orientation of the interest point.

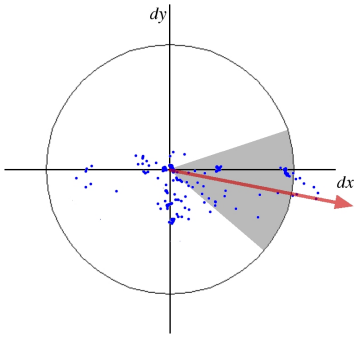

Figure 4.1c. Orientation assignment with a sliding window of $\frac{\pi}{3}$

4.2. High level architecture

Figure 4.2 shows the high-level architecture of our system. The input images are externally converted to grayscale (as the SURF algorithm specifies). Those images are 640x480 as requested by the majority of the existing machine vision systems. The core of our system consists of four main modules as shown in Figure 4.2. The first one computes the integral representation of the input image while the second locates the interest points. The third module is divided into three sub-modules: The first computes the determinant response map, the second implements the non-maximal suppression scheme described in the last section, while the third executes the interest point localization task of the SURF algorithm. The last stage actually computes the orientation of the reported interest points.

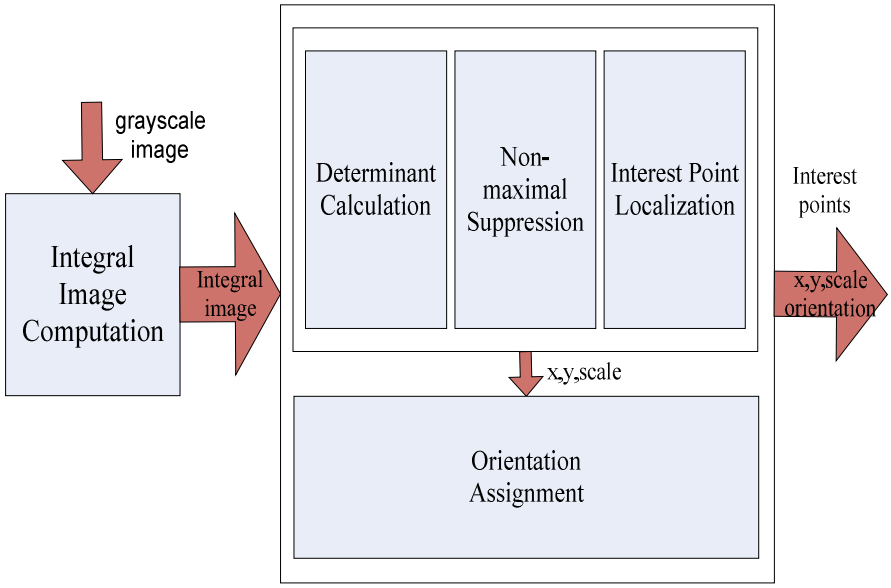

Figure 4.2. High-level Architecture of feature detector

4.2.1. Integral Image Module

The integral image module produces the integral representation of the input grayscale image. In the software implementation the actual pixel values are normalized by a factor of 255. However, in our hardware system it was more convenient to perform the normalization by a factor of 256 as this division is effectively a bit shifting. As we demonstrate in the performance section in the next chapter we do not lose any significant accuracy due to this slight change in the normalization factor.

4.2.2. Determinant Module

The determinant module computes the determinant of each pixel for all the scales. It consists of the retrieve parameters module, the Dxx,Dyy,Dxy Computation Module and the Determinant Calculation module. Figure 4.3 demonstrates the Determinant Module's micro-organization.

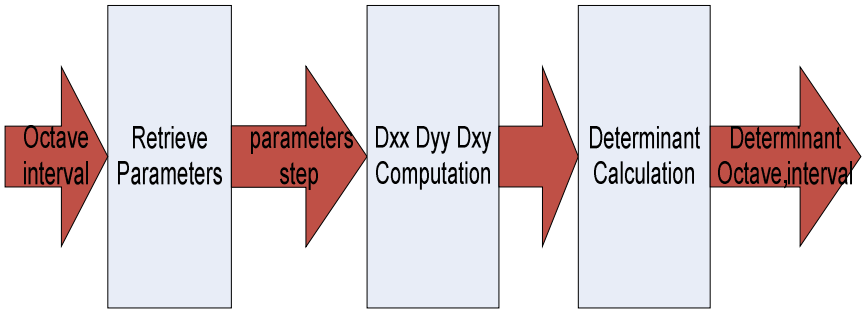

Figure 4.3. Determinant Module

Retrieve Parameters Module: The scale-space concept is analytically described in Section 4.1. In order to perform all the employed calculations we need several parameters which are retrieved by this module, which essentially utilizes a number of look-up tables.

Dxx,Dyy,Dxy Computation Module: The Basic unit of this module is the Box-integral function unit. This unit performs four memory accesses, three additions/subtractions and it calculates the intensity of a rectangular area. In total, eight rectangular area calculations are required, two for Dxx, two for Dyy and four for Dxy and all of them are performed in parallel.

Determinant Calculation Module: This module mainly implements equation (7) which is proposed by Bay [BAY et al. 2006] and computes the determinant of the approximated Hessian matrix. Using the results from the previous module and some multiplication units, we compute the determinant for a specific pixel location in a specific scale.

$$\det(H_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2 \quad (7)$$

As analytically described in [BAY et al. 2006], the number of detected points decays quickly for each octave and thus we just need three octaves for describing all the important information of an image. The determinants, as described in Section 4.1 (Surf algorithm), are computed for certain pixels of the image based on a sampling interval.

In the software implementation the determinant response map is stored into a matrix with total size equal to *image size * number of octaves* number of intervals*. In particular, the scale-space is divided into a number of octaves, and each octave is divided into four intervals. The Non-maximal suppression module and the interest point localization one are using the determinants of each octave as well as those four intervals.

In order to allow the pipelining execution of the non-maximal suppression function with the determinant calculation we altered the algorithm in the following way: Instead of calculating the determinants for every pixel at each octave (and for all intervals) and then starting the non-maximal suppression procedure, we compute the determinants for five rows and then we pass the results to the non-maximal suppression module; obviously we then continue with the next 5 rows etc. We use three distinct structures in order to store the determinants of the five rows as demonstrated in figure 4.4. Our choice for the number of rows and for the number of structures is explained in subsection 4.2.3 (Store Determinant Module).

4.2.3. Non-maximal Suppression Module

The non-maximal suppression module locates the interest points in all different scales. The module consists of three main cores that locate and localize the interest points both in scale and in space. Figure 4.5 presents the overview of this module.

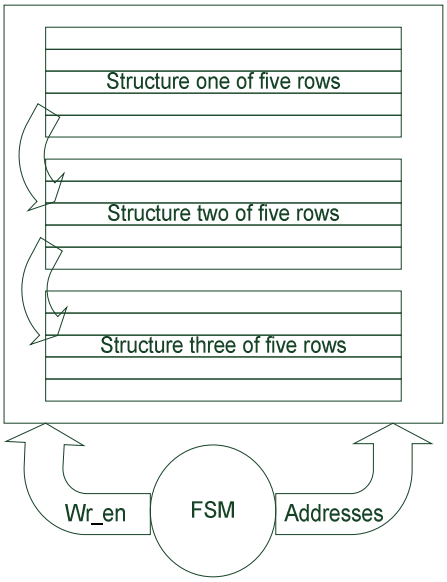

Figure 4.4. Determinant calculation and non-maximal suppression scheme

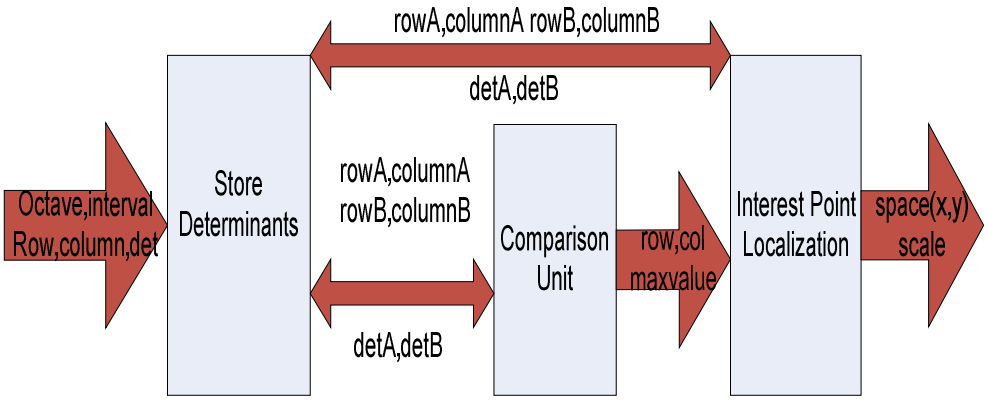

Figure 4.5. Non-maximal suppression overview

Store Determinant Module: This module computes the determinants of five consecutive rows; three of those rows are needed in order to find the maximum determinant in a 3x3 region while the two neighboring to this region rows are needed in order to find whether the value from the previous comparisons is the maximum value detected, among all the neighboring pixels determinants, or we have to select the newly created comparison value.

Comparison Unit Module: This module performs the non-maximal suppression task and reports the candidate interest points as described in Section 4.1; effectively this sub-system performs certain comparisons so as to come with a candidate set.

4.2.4. Interest Point Localization Module

In order to compute the difference between the neighboring pixels the algorithm employs a certain interpolation method as described in Section 4.1. This method utilizes two matrices consisting of the finite differences of neighboring pixels in space and in scale. The actual result of the module is a 3x1 matrix containing certain “factors”; those factors, depending on their values as described in Section 4.1, may be used to adjust the pixel locations and their scale. Figure 4.6 provides an overview of the high-level architecture of this module.

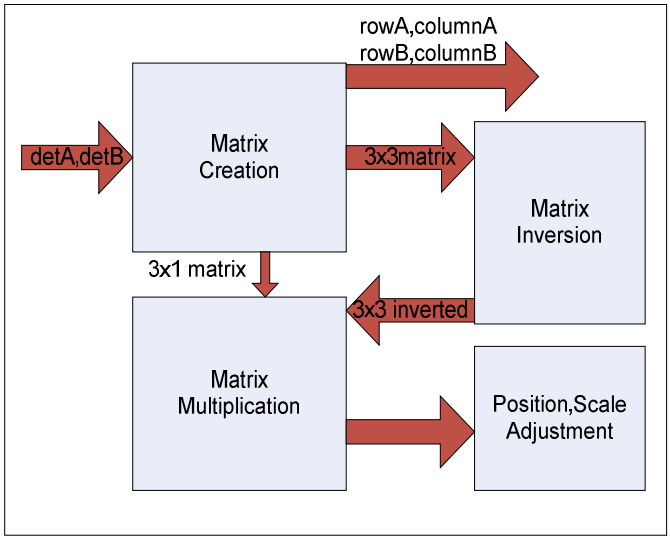

Figure 4.6. Interest Point Localization overview

4.2.5. Orientation Assignment Module

This module implements the orientation assignment task for each interest point. Based on the SURF algorithm, there is a single vector in which the filter responses (HaarX, HaarY) of all the pixels around a detected interest point are stored. In particular, this vector contains the Haar responses for 109 pixels which are located around a certain interest point and within a specified radius (as described in Section 4.1).

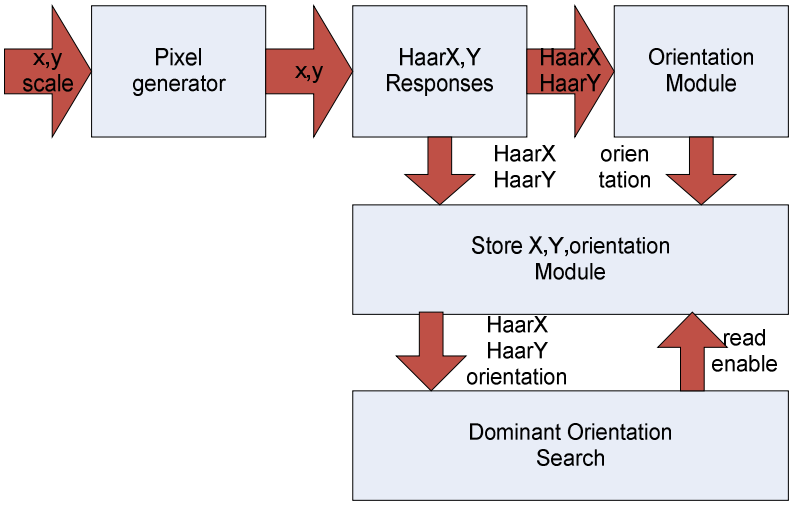

Figure 4.7. Orientation Assignment module overview

During this process each pixel of the vector is checked in order to determine whether it belongs to a specific subspace according to a certain angle. Therefore, the software accesses this single vector 109 times so as to search for all the possible angles. In order to be able to parallelize this important task we have partitioned the vector into seven parts which are stored into seven distinct memory modules of 16 lines each. Therefore, our implementation can search for the angles of seven vector responses in parallel in each clock cycle. The above organization is presented in figure 4.7

4.3. Hardware Implementation

In order to implement the architecture of Section 4.2 in a very efficient manner we have utilized certain techniques which are listed below.

4.3.1. Image prefetching

In order to be able to support video at relatively high rates we pipelined the image loading with the image processing tasks. Therefore, we have two distinct memory banks and while we store the input image in one of them we process the previously stored image which has already been placed in the other bank. When the processing (which takes much longer than the image loading) is completed the two banks change roles.

4.3.2. Arithmetic

In order to improve the performance we have used fixed point arithmetic in the majority of our modules, namely the Integral image processing one, the determinant computation, the non-maximal suppression module and the orientation subsystem; as we clearly demonstrate in the performance chapter (Subsection 7.1.1) the use of fixed-point arithmetic in those modules did not reduce significantly the accuracy of the system. However, in the Interest point localization module we use single precision floating-point arithmetic (just as the reference open-source software). We could not avoid using such arithmetic since the matrix values in our greyscale image belong to the [0, 1) space and as a result the determinant of the 3x3 matrix is a very small and varying number; at the same time the computed inverted determinant is a very large number.

4.3.3. Memory organization

In order to achieve better utilization of the on-chip block memory (i.e. BRAM) and higher performance we partitioned the integral image (i.e. each of the two banks described in 4.3.1) into five smaller parts. Each part has 2^{16} (65536) rows and stores 640x100 (64000) integrated pixels except of the last one which stores only 640x80 pixels. Those five BRAM-based memory blocks support simultaneous processing of the image by more than one detection cores given that the five parts of the integral

image do not overlap. By carefully analyzing the box integral function, which is the only one accessing the raw image, we identified that the addresses of any two continuous accesses always differ by an odd number. Thus, by placing the odd and the even rows in different memories we can always perform two consecutive memory accesses in one cycle. Since the BRAMs we have utilized are dual-ported, our module accesses four different rows simultaneously. We have implemented a dual-core detection module and a single core orientation one since each detection core has slightly less than half the bandwidth of the orientation core; as table 7.1 (Subsection 7.1.2) clearly demonstrates in this way the proposed pipeline organization is almost perfectly balanced.

4.4. Hardware Cost

Our system has been implemented in a Xilinx Virtex 5 XC5VFX130T FPGA. The following tables demonstrate the hardware cost of our design. Table 4.1 and 4.2 list the resources covered by each of the modules. The numbers in parenthesis indicate the percentage of the total resources of that kind available in the selected device. As clearly demonstrated in Table 4.3 our system can easily fit in this memory-reach device while it is clocked at 200MHz.

Table 4.1. Resource utilization of Interest point detection sub-modules

Module	Slice Registers	Slice LUTs	LUT-FF pairs	Block RAM	DSP48Es
Integral Image	88 (0%)	5880 (7%)	5947 (7%)	217(72%)	0 (0%)
Retrieve Parameters	10 (0%)	94 (0%)	8 (0%)	0 (0%)	0 (0%)
Dxx Dyy Dxy Computation	51 (0%)	281 (0%)	70 (0%)	0 (0%)	0 (0%)
Det Calculation	170 (0%)	246 (0%)	82 (0%)	0 (0%)	13 (4%)
Store Determinants	14 (0%)	992 (1%)	35 (3%)	24 (8%)	0 (0%)
Comparison Unit	252 (0%)	882 (0%)	263 (15%)	0 (0%)	0 (0%)
I_Point localization	3888(4%)	3760 (4%)	2355 (2%)	0 (0%)	12 (3%)

Table 4.2. Resource utilization of Orientation assignment sub-modules

Module	Slice Registers	Slice LUTs	LUT-FF pairs	Block RAM	DSP48Es
Orientation	3208 (4%)	3349 (4%)	3623 (4%)	0 (0%)	0 (0%)
Store X,Y, Orientation + Dominant Orientation	836 (1%)	5204 (6%)	5209 (6%)	21 (8%)	26 (8%)
HaarXY	141(1%)	139 (1%)	197 (1%)	0 (0%)	0 (0%)
Pixel Generator	23 (1%)	52 (1%)	52 (1%)	0 (0%)	0(0%)

Table 4.3. Resource utilization of Interest point detection and Orientation

Module	Slice Registers	Slice LUTs	LUT-FF pairs	Block RAM	DSP48Es
Dual core Interest point detection	11457 (13%)	13272 (16%)	17849 (21%)	271 (90%)	50 (15%)
Orientation assignment	6578 (8%)	11808 (14%)	13092 (15%)	244 (81%)	18 (5%)

Chapter 5

RFCH Embedded System

In this chapter we give an overview of the RFCH algorithm and we also present in detail a novel embedded architecture accelerating the core of the algorithm. We also describe a verification platform of this architecture and outline the hardware cost when implemented in certain FPGA devices.

5.1. The RFCH algorithm

5.1.1. 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, the histograms produced are called regular color histograms.

A Receptive Field Cooccurrence Histogram (RFCH) is able to capture most 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. In this way only pixel pairs located within a maximum certain distance, dmax, are considered. Thus, the histogram represents not only how common a certain descriptor response is in the image but also how often certain combinations of descriptor responses occur close to each other. In other words, an RFCH is a representation of how often pairs of certain filter responses and colors lie close to each other in the image. This results in a representation of the image in which most of the geometric information is preserved thus allowing for more accurate object recognition. Figure 5.1 below presents the concept of the cooccurrence histogram, of a 3bit (8-color) greyscale image, where we search for co-occurrences from left to right with dmax = 1.

5.1.2. Receptive Field Cooccurrence Histogram for Object Detection

One of the main advantages of this algorithm is that it can work with numerous different types of image descriptors such as Color, Gradient magnitude, Laplacian, Gabor as well as any mixture of them. As it is has been proved in [EKVALL et al. 2005] for object recognition the optimal choice is to utilize rotationally invariant image descriptors such as Color, Gradient magnitude and Laplacian descriptors and the actual choice can depend, among others, on the image characteristics.

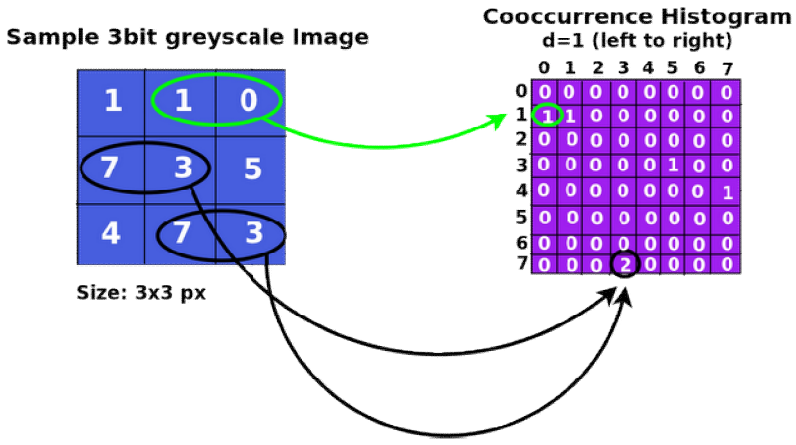

Figure 5.1. High level view of RFCH algorithm

5.1.3. Image Quantization

When utilizing histograms in the recognition process, the computational complexity of the algorithm increases exponentially with the dimensions of the histogram. In order to alleviate this problem the algorithm firstly clusters the input data, so as to reduce the histogram dimensions. Hence, by altering the number of clusters the histogram size may be controlled. The cluster centers (N) have a dimensionality equal to the number of image descriptors used. The adopted algorithm is using the K-Means clustering algorithm [MACQUEEN et al. 1967] for the dimension reduction. In particular, after quantization, each object ends up with its own cluster scheme which is used together with the RFCH which has been calculated on the quantized training image. When searching for a certain object in a scene, the whole image is quantized with the cluster scheme that has been applied in the quantization of this search object.

5.1.4. RFCH-based object detection phase

After the clustering step, the algorithm creates the object's cooccurrence histograms in 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 at any given instance. In each scan the RFCH of the window is compared with the object's RFCH.

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

$$\mu(\mathrm{h}_1, \mathrm{h}_2) = \sum_{n=1}^{\mathrm{N}^2} \min\left(\mathrm{h}_1[\mathrm{n}], \mathrm{h}_2[\mathrm{n}]\right)$$

where $h_i[n]$ denotes the frequency of receptive field combinations in each discrete interval (bin) n for image i, when quantized into N cluster centers. The higher the value of the $\mu(\mathrm{h}_1, \mathrm{h}_2)$ the better the match between the histograms, and as a result, the better the match between the search object and this specific part of the image.

As a summary the algorithm works in two phases and performs the following steps in order to detect a certain object in an image:

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 image window (for all possible image windows)
- Match Object and Image RFCH with histogram intersection (for all windows)
- Report the best match

5.2. Hardware/Software Partitioning

In order to create an efficient embedded system, we first analyzed the RFCH application so as to be able to perform the optimal hardware software partitioning. In order to profile the software implementation of the algorithm we have used Intel's VTune Amplifier XE 2011 [VTUNE]. The profiling was performed on an Intel SU7300 Dual Core ULV CPU working at 1.3GHz since this is a low-power CPU found in embedded multimedia systems (as for example [MS-9A35]). The same profiling results were also produced when executing the same code on an ARM placed in a Gumstix device [GUMSTIX]. All of our experiments were conducted using the original optimized software provided by the inventors of the underlying algorithm [EKVALL et al. 2005] along with images from the most widely used Image Database, the CVAP Object Detection Image Database [CVAP], which we have rescaled to 640x480.

After running various tests combining different scenes and objects we concluded that, functions CalculateClusters(), ClusterFeatures() and CalculateRFCH() are taking 97.8% in average (and at least 96%) of the total execution time. By making the above 3 functions faster, we can significantly improve the performance of the overall algorithm; according to Amdahl's Law the maximum theoretical speedup in that case is 45x. We have also analyzed the interconnection needed if those functions are implemented in hardware and the rest of the functions for Feature Extraction (i.e. Create Image Gauss, Create BW Image etc) and Histogram Intersection (i.e. MatchRFCHs) are executed in the CPU and found it to be minimal as described in the next section. In particular, even though CalculateRFCH takes only 8% of the total time we implemented it in hardware so as to minimize the data transactions between our hardware modules and the embedded CPU.

Another important reason for implementing the Feature Extraction as well as the Histogram Intersection Algorithms in software is that it allows us to easily change those parts of the algorithm depending on the image characteristics (e.g. change the actual descriptor used) thus heavily increasing the applicability as well as the accuracy of the end system. Before we have actually implemented those functions in hardware, and in order to be able to fully dimension the problem, we have also measured the computational complexity of those 3 functions.

CalculateClusters: This function implements an iterative version of the K-Means algorithm, and it has been identified as the major hot-spot during the profiling procedure. 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.

ClusterFeatures: This function is responsible for the quantization of the image according to the pre-calculated cluster centers. The function has a complexity of O(nfN). The function takes as input the Feature Array and the Cluster Point Array and produces the Binned Image Array.

CalculateRFCH: The complexity of this function is approximately $O(nd^2)$, where *n* is the Image Size and *d* is the maximum distance (d_{max}).

5.3. RFCH System Architecture

Moving to the implementation of the previously identified hot-spots of the presented scheme, we have decided to use a Xilinx Virtex-6 FPGA, which resides on the ML605 Xilinx Evaluation Board [UG534]. Those designs have been implemented manually in VHDL and we have synthesized, mapped, placed and routed them using Xilinx ISE 14.5.

The main concept of our approach is that the three HW accelerated functions are placed one next to the other in such a way so as to minimize the data being sent from and to the CPU executing the rest of the functions in software. By adopting this approach we don't need to have 3 independent data transactions to the reconfigurable fabric which will trigger a significant communication overhead. In the proposed architecture, demonstrated in Figure 5.2 we transfer data from the CPU to the FPGA practically only when loading the Feature memory; then our hardware modules process those data until the complete image slice is fully processed. The loading time for the feature memory is very low and up to about 0.05msec for the 640 x 480 images (for a typical 100MHz bus as in [PCI_BUS]) while the software

processing does not need more than 1msec at any experiment conducted. Moreover, as it is demonstrated in the next section, we have utilized a double buffering scheme in order to pipeline the loading and software processing time of the different slices of every image with the actual hardware processing of them. The write back time is negligible as the only thing we need to transfer is the RFCH result which is a 80x80x11 bits datum and this is only needed once for each complete image. Figure 5.2 also demonstrates the data flow through the implemented modules.

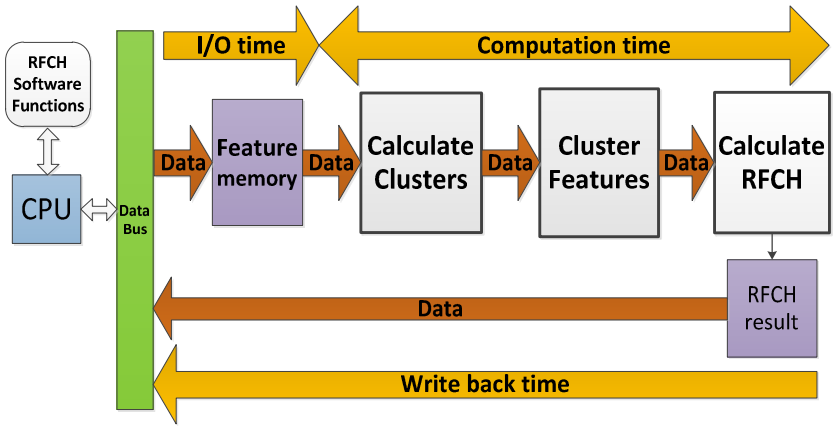

Figure 5.2. High Level Architecture

5.3.1. Calculate Clusters Module

The CalculateClusters function is responsible for the clustering of the features array, and it mainly implements an iterative version of the K-Means algorithm. It is applied during the training phase of the algorithm and it works in 3 distinct phases: Phase 1 and 2 perform the actual calculations while phase 3 updates the cluster centers. The overall micro-architecture of this module is demonstrated in Figure 5.3. Processing Unit A (PUA) is calculating the cluster centers and it utilizes 16 cores. Each core can perform the necessary processing on a small image slice of size 640 x 2 (which consists actually of 2 lines of the feature image). The whole feature image (640x480 x7 features-8bit) is pre-segmented by the software and it is sent in slices (i.e. 32 lines per processing cycle) to the hardware module. As a result the Feature Array/Memory utilized in each core is 640 x 2 x 7 bytes, and with the proposed configuration, each module can process 16 feature image blocks (640 x 2 x 7)

simultaneously. In each processing core we have to execute a critical multiply-and-accumulate (MAC) operation; in order to speed up this function we have utilized a pipelined Digital Signal Processor (DSP) built-in core, found in those Xilinx Virtex6 devices.

When the processing is completed, Processing Unit B (PUB) sums all the intermediate results produced by the 16 cores. When the sum is fully calculated, PUB triggers Processing Unit C which is responsible for updating the cluster centers as well as the clusterPoint array; the latter is also split into 16 slices. The clusterPoint array holds the calculated clusters information needed for the clusterFeatures module as it is described in the next paragraph.

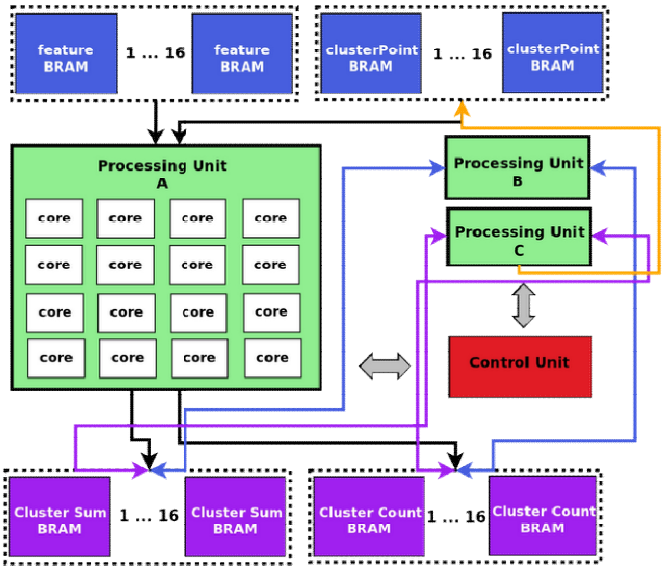

Figure 5.3. Calculate Cluster HW implementation

5.3.2. Cluster Features Module

This module implements the image quantization task and it also utilizes 16 parallel cores. The inputs of this module are a) the feature array and b) the clusterPoint array calculated from the CalculateClusters module. Concerning how the data are decomposed and processed in parallel the exact same technique with the one described in the last section is applied.

In particular, we used images of sizes 640 x 480 and utilized 7 distinct features. This means that our feature array is equal to 640x480x7(x8bits) = 2.15 Mbytes which cannot fit in the on-chip RAM. In order to be able to load the feature array on-chip,

while also processing a sub-part of it, we have split it in 15 slices; in this way we load a certain slice to the FPGA while simultaneously we process the previous slice. Those 15 slices are of size 2.15/15 = 0.13 MB each. In that way in order to process the whole feature array, we have to process 15 slices.

Then we split further the on-chip slice into 16 blocks with size (640x480x7)/240 = 8.75 KB each and then we pass each block to a distinct processing core (i.e we utilize all 16 parallel processing cores in order to process one slice). The above procedure is depicted in Figure 5.4

The high level architecture of the cluster feature module is demonstrated in Figure 5.5. Each slice of the feature image array (640 x 32 x 7) is fitted in 16 distinct feature Memories (RAMs). Each Feature RAM can hold a block equal to 640 x 2 x 7 (using 8-bit color). The clusterPoint array of size 7 x 80 is initially loaded into the 16 distinct clusterPoint RAMs each of size 7 x 80x 8 bits. The actual processing comprises of each core quantizing an image block of 640 x 2 pixels (i.e 2 lines) as follows: The first core quantizes 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 thus fully utilizing the 16 distinct processing cores. Each core also has a dedicated BRAM for storing the results. This BRAM is the binnedImage memory with a total size of 640 x 2 x 8 bits. Each core performs the same MAC operation, as in the CalculateClusters case, so we also utilize here a fully pipelined built-in DSP core; in total we need 16 DSP slices to support this module.

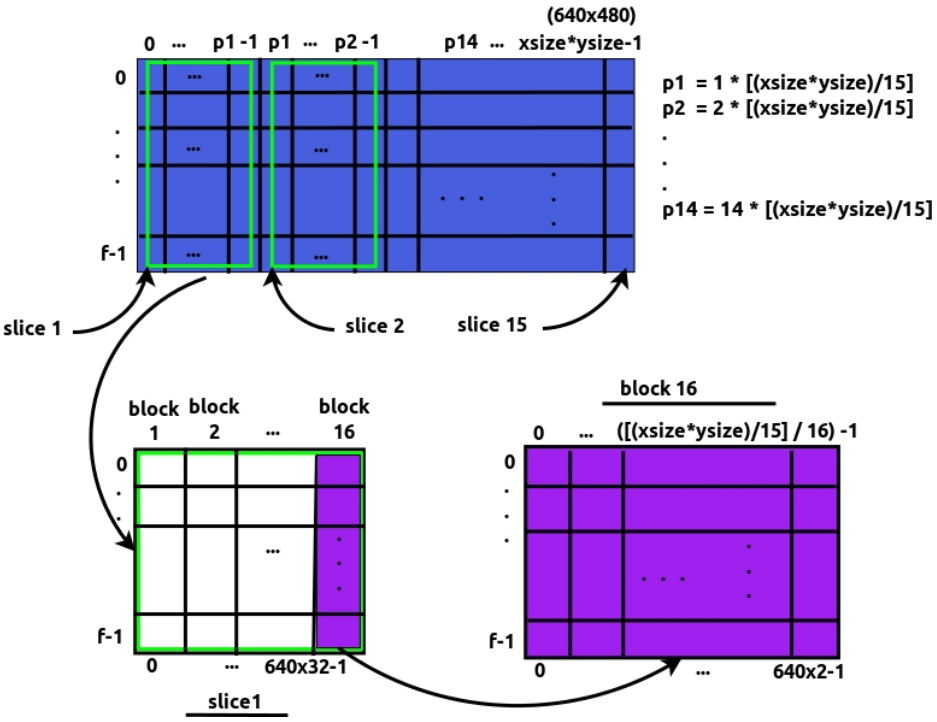

Figure 5.4. Feature Array decomposition

When all the cores have completed the corresponding processing, an image slice has been fully quantized and the results reside in the 16 binnedImage RAMs. The control unit for the ClusterFeatures module is quite simple. It just monitors when all the cores have finished their processing and then it loads the next block.

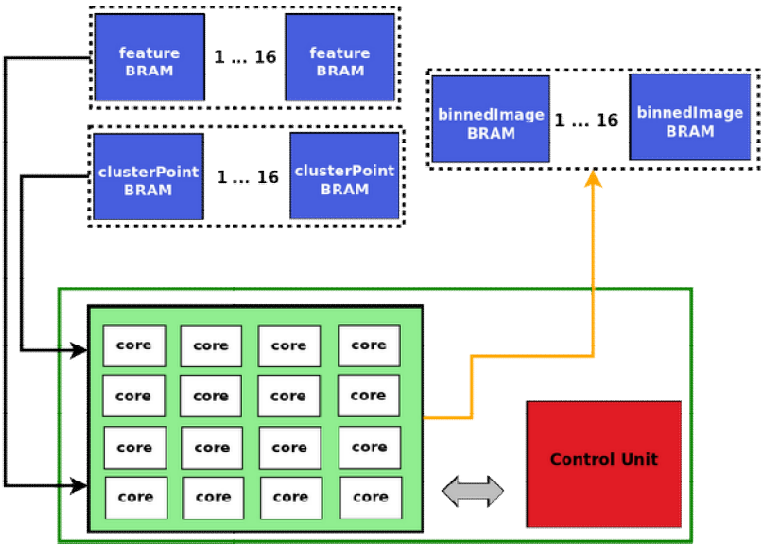

Figure 5.5. Cluster Features module

5.3.3. Calculate RFCH Module

This module calculates the required receptive fields' cooccurrence histograms. The overall architecture of the module is presented in Figure 5.6. The module utilizes 8 processing cores as it is less demanding, in terms of processing time, than the other two modules.

The RFCH is calculated based on the BinnedImage data. The binnedImage array is of size 640 x 480 and, as we presented in the last section, it is the quantized version of the image. In order to calculate the RFCH for each binnedImage slice of size 640 x 32 we need to process the data coming from two continuous blocks; the additional block/slice is needed since we need 4 extra lines in order to serve the dmax=4 condition (i.e. each core should look up to 4 lines ahead thus utilizing the data of the next slice). The last block of the current slice is paired with the first block of the next slice in order to keep the dmax condition valid between image slices. The above procedure is shown in the Figure 5.7.

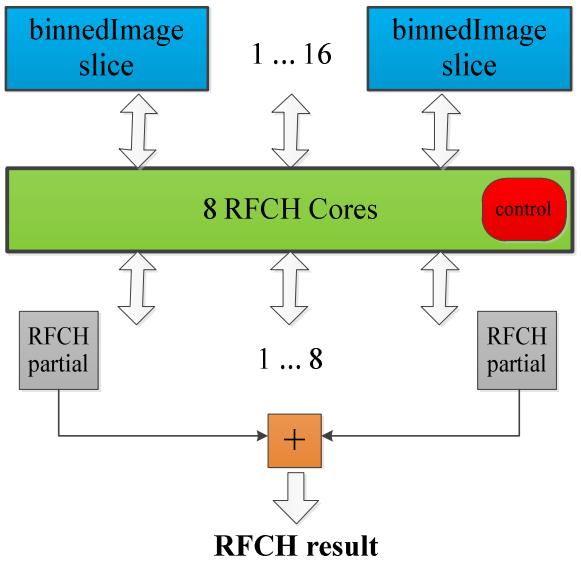

Figure 5.6. Calculate RFCH module organization

Each of the eight cores calculates the cooccurrences of a binnedImage block of size 640x4. This means that the 8 cores together can calculate the cooccurrences of a binnedImage slice equal to 640x32. As previously mentioned, in the current version

of the system, the cooccurrenses are calculated based on a specific value of dmax (dmax = 4) which gives very high accuracy as described in [EKVALL et al. 2005]. If, for any reason, we decide to use a larger value for dmax, we will have to utilize more memory since the RAM blocks needed, in the presented architecture, are equal to 2dmax. Each processing core maintains a dedicated memory block for its output in which a partial RFCH is stored; the required memory size is 80x80 x 11 bits. In order to calculate the RFCH of the whole binnedImage of size 640 x 480 we have to process 15 binnedImage slices of 640x32 each and update the corresponding memories. After we process all the slices we store the results in the corresponding 8 RFCH memories; then those stored results are summed in order to form the final RFCH for the image and this is the end-result sent to the CPU for further processing.

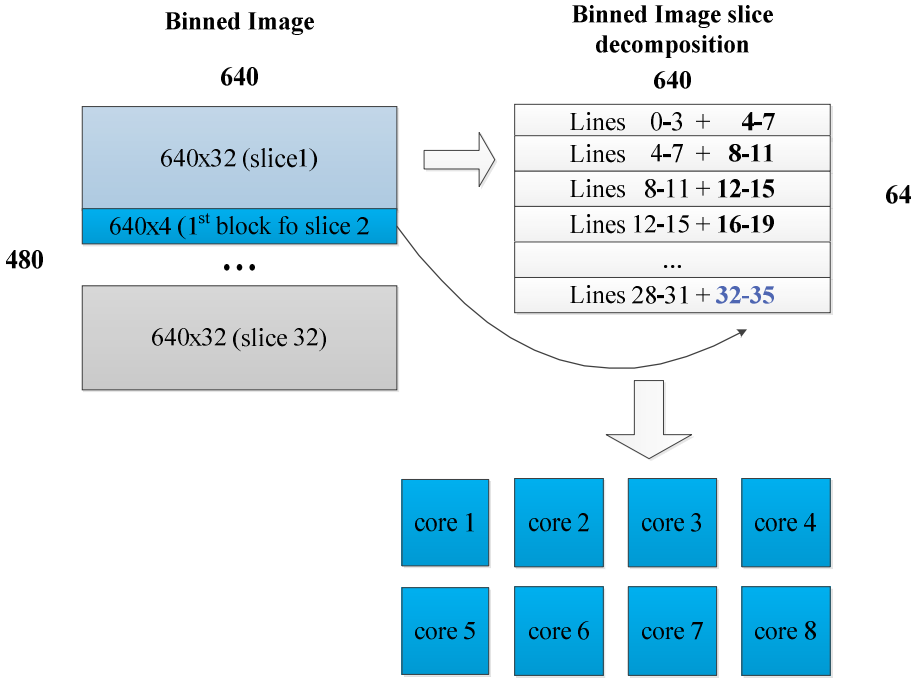

Figure 5.7. Binned Image Array Decomposition

The control of this module is quite complicated since it contains various termination criteria and jump conditions involved in the cooccurrence histogram computation. In

particular it is implemented as a 14-state Finite State Machine and it has been fine-tuned in order to achieve the maximum possible clock rate.

5.3.4. Verification of the complete System

In order to verify the correct functionality of our approach and mainly the feasibility of our complete system being implemented in a single chip, we integrated all the hardware modules as well as the software code in an embedded design platform based on a Virtex 6 FPGA board [UG534]. For this proof of concept we have selected the on-chip Xilinx Microblaze [UG081] softcore processor for the software execution, connected to the AXI / AMBA bus. We used the AXI Block Ram Controller provided by the Xilinx EDK 14.5 environment to make all the necessary data transfers between the CPU and the reconfigurable modules. The controller occupies the first port of a dual port configuration RAM leaving the second port to run on the custom-made hardware side at a different clock domain (i.e the Frequency of the Microblaze CPU was fixed at 150 MHZ while our custom hardware works at 350MHz).

We also used the AXI IP Interface modules (IPIF) to utilize user specific software accessible registers. In that way we were able to control and monitor the status of our custom accelerator from the software side. In this particular platform a soft core CPU was the only choice as the Virtex 6 family is not equipped with any hard core CPUs (like the ARM found in the already announced Virtex-7 FPGA under the codename Zynq-7000 [DS190]). Unfortunately the Microblaze is performing poorly when implementing the software functions and as a result it degrades the overall performance of the system (to an overall speedup when compared with the reference Intel CPU, of about 3.5x); thus we used it only for verification and feasibility purposes. We also used the compact flash peripheral in order to load the test images and verify the results. Figure shows an overview of the proof-of-concept embedded platform.

The Microblaze talks to an external DDR SDRAM in which we place images from a well known image database [CVAP] as well as real-world multimedia data (i.e. real-world videos). The Microblaze executes the part of the RFCH algorithm that we mapped to the CPU, when performing the hardware/software partitioning, while it

additionally controls all the system's peripherals. In this initial single-chip approach the functions that were implemented in hardware have been called by the corresponding software drivers independently (i.e. 3 different calls, one for each hardware module) since that allowed for much faster and easier debugging of the hardware cores. The RFCH controller is the hardware module supervising the interconnection between the Microblaze and the RFCH hardware accelerator, and it is attached as an AXI slave IPIF module. This module is basically an AXI4lite slave [DS768], which provides a bi-directional control/status interface between the RFCH hardware accelerator and the AXI bus. After executing the complete algorithm in the specified FPGA platform we compared the end results residing in the DRAM with those triggered by the pure software solutions and they were identical.

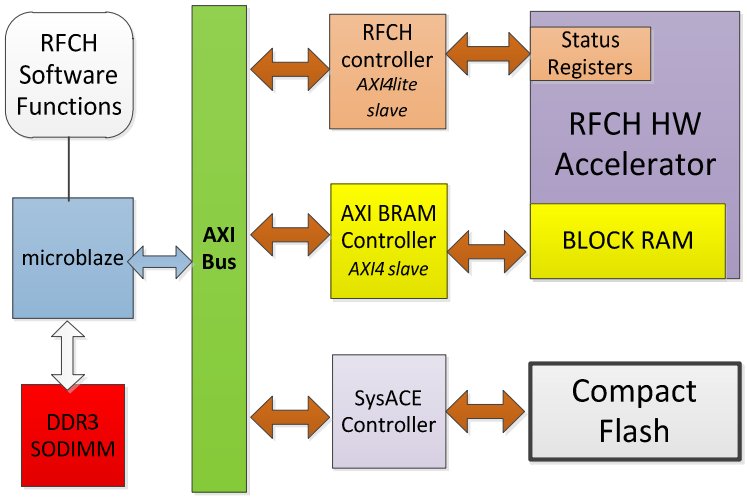

Figure 5.8. Micro-Architecture of our proof of concept single-chip approach

It should also be stressed that in numerous current embedded devices the image size is 640 x 480. As a result our prototype hardware implementation, as well as the performance results of the relevant chapter, are all based on that image size. However, our system can seamlessly (i.e. with a simple change in a couple of control modules) process any images up to 1920 x 1080 (High Definition-HD) pixels; and this is an analysis that will soon been incorporated even in mobile embedded multimedia

systems. The only difference will be that those HD images will consist of more slices and therefore more time will be needed in order to process a complete such image.

5.4. Hardware Implementation Cost

The following table shows the utilization of each function on both a Virtex-6 VLX75T and a Virtex-6 VLX130T device. The total utilization of the three functions leaves room for more cores to be implemented in both devices if needed.

Table 5.1. Hardware Cost on a Virtex-6 VLX75T device

(16 Cores C.Clusters and Cluster Features, 8 cores for Calculate RFCH)

Module	Slice Registers	Slice LUT	Block RAM	DSP Slices
Calculate Clusters	26,082/93,120 28%	23,691/46560 50%	112/312 35%	17/288 5%
Cluster Features	4,176/93,120 4%	5,840/46,560 12%	112/312 35%	16/288 5%
Calculate RFCH	1,066/93,120 1%	3,465/46,560 7%	16/312 5%	0/288 0%
Total	31,324/93,120 33%	32,996/46,560 70%	240/312 76%	33/288 11%

As demonstrated in Table 5.1 by using small image slices (as described in subsection 5.3.1) we minimized the RAM usage while exploiting high levels of parallelization. The small footprint of our design, in terms of Slice LUTs, is giving as the choice to double the cores in the current design and assign 1 image line per core with neither increasing the amount of utilized Block RAMs nor the control complexity. The utilization for this system is demonstrated in Table 5.2. This approach almost doubles the performance of our system, as the performance chapter clearly

demonstrates, while making the LUT-to-BRAM factor very close to the LUT-to-BRAM factor of the available resources in the middle Virtex-6 FPGAs.

Table 5.2 . Hardware Cost on a Virtex-6 VLX130T device

(32 Cores C.Clusters and Cluster Features, 8 cores for Calculate RFCH)

Module	Slice Registers	Slice LUT	Block RAM	DSP Slices
Calculate Clusters	5,4642/160,000 34%	48,121/80,000 60%	112/528 21%	32/480 6%
Cluster Features	8,769/160,000 5%	11,986/80,000 15%	112/528 21%	32/480 6%
Calculate RFCH	1,066/160,000 1%	3,465/80,000 4%	16/528 3%	0/480 (0%)
Total	64,477/160,000 40%	63,572/80,000 79%	240/528 45%	64/480 13%

5.5. Techniques for further acceleration of the architecture

Since our system can process different images totally independently of one another, by adding more memory and logic resources we will trigger a further speedup and we expect that the performance will scale linearly with the device utilization. In order to further support this case we have placed two instances of the system described in the last paragraph to a Virtex-6 VLX240T (the total utilization for both BRAMs and Slices was close to 95%) and since the I/O was not the bottleneck, we managed to eventually double the supported bandwidth. As will be explained further in the performance chapter , even if the loading time is doubled due to the sharing of the bus bandwidth among the two cores it is still far less than the processing time. This fact in addition with a double buffering scheme eliminates any I/O overhead. However, since we mainly focus on low-energy, relatively low-cost multimedia embedded applications, the performance results demonstrated in the performance

chapter, cover the implementations of our system on both a Virtex-6 VLX75T and a Virtex-6 VLX130T low-cost devices connected to a low-cost ARM core.

Since our aim was to implement a low-power embedded multimedia system, special care has been taken so as to reduce the overall power consumption of our novel device. In particular, we implemented, in each module, numerous parallel simple cores working at smaller speeds instead of creating complex cores with large and complicated control working at higher speeds. In order to achieve that we decomposed the actual processing, in each module, in smaller parts; special care has been taken so to reduce the, triggered by this decomposition, overhead in terms of memory repetition and/or memory reads/writes. The resulted reduction in the energy consumption by adopting this technique was up to 17%.

We have also introduced the pipeline of Figure 5.2 so as to minimize the data moving to/from the CPU; for example, by implementing the "Calculate RFCH" module in hardware we heavily decreased the intercommunication needed with the CPU and that resulted to a 12% reduction in the overall power consumption.

Additionally, with the aim of the FPGA design tools, we have placed the logic as close as possible to the Memory Banks therefore minimizing the necessary routing while we have also hand-designed from scratch all the processing cores (so as to eliminate any unnecessary silicon), instead of using the ready-made IP cores provided by third parties; those techniques reduced the overall power consumption by an additional 11%.

Moreover, we have utilized a double buffering scheme, as demonstrated in the performance chapter (Subsection 7.2), which, together with the introduced pipeline of Figure 5.2, allow us to utilize all the hardware resources at any given time. Since, certain hardware units consume power even when they are idle (i.e. do not produce any useful results), by effectively removing any idle states we decreased the actual energy consumed when the complete application is executed by a further 20%.

Chapter 6

OpenTLD Embedded system

In this chapter we present a long-term object tracking algorithm, the OpenTLD. We give a brief overview of its architecture and also describe a very efficient way to implement an embedded system, without restricting its internal parameters or the application-specific customization that the original software version supports.

6.1. The OpenTLD algorithm

6.1.1. Introduction

The OpenTLD algorithm [KALAL et al. 2009] performs accurate long-term object tracking in real time. The OpenTLD's classifier which is used in the detection task utilizes the randomized forest concept [BREIMAN 2001]. More specifically, the features used are randomized in terms of position, scale and aspect ratio. As proven by various profiling tests we have performed in an Intel state-of-the-art CPU, the main task, which is the one performing the actual detection by involving the Random Forest Classifiers, spends about 90% of its execution time in memory related tasks (memory accesses and addressing); this proves that the OpenTLD scheme is a very memory intensive one. As a reference software implementation we used the one in [BPTLD] which has been the first published implementation of this scheme in C++.

6.1.2. Random forest

The randomized forest approach which yields high classification accuracy is based on an ensemble of trees which vote for the most popular object. In order to create these ensembles, random vectors are generated and those vectors govern the growth of each tree in the ensemble. The random forests consist of a combination of tree predictors such that each tree depends on the values of a random vector sampled independently while the same distribution is applied for all the trees in the forest [BREIMAN 2001]. As a result the classification algorithms utilizing this approach are describing each object/class patch by a number of local features; in the

case of vision algorithms this includes position, scale and aspect ratio which are all generated at random.

A random forest classifier usually utilizes several decision trees (e.g 15-20) and each decision tree is traversed on a number of features (e.g 8-12) [KALAL et. al 2009]. The leaf-nodes of each tree specify the probability of an object match and the probabilities of all the trees are averaged so as to yield a final probability. As all decision trees are sampling the same integral image at the feature extraction phase, the memory bandwidth is the most critical factor in terms of performance (i.e. it is a memory bound problem).

The integral image is an intermediate interpretation of an actual image and it is used very efficiently in the Viola and Jones framework [VIOLA et al. 2001] which is very often utilized in vision applications. Viola and Jones in their work showed that by applying a large number of simple rectangular filters in an image it is possible to build a robust feature-based description of it. Those rectangular features are rather primitive features when compared to alternatives such as steerable filters introduced and utilized in other algorithms. They do however provide a rich image representation which supports effective classification and also they can be evaluated in constant time which gives them a considerable speed advantage over their more sophisticated relatives.

Figure 6.1 demonstrates the difficulty in exploiting fine grain parallelism (i.e. at the forest access level) in Random Forest classifiers even if a distributed memory scheme is utilized. This is because the randomized sampling of the integral image may compromise the performance since many or even all of the memory accesses needed for each feature extraction can refer to the same memory block. In other words we cannot efficiently and safely distribute the memory accesses across the different memory blocks during the randomization procedure.

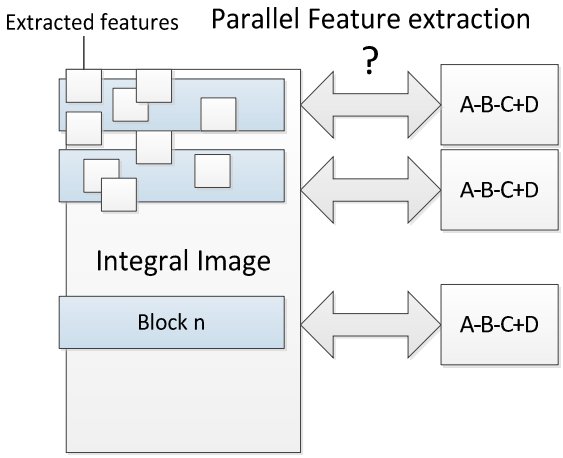

Figure 6.1. Feature extraction in Viola and Jones framework with randomized sampling

On the other hand the Random Forest Classification method can be conceptually parallelized since its learning algorithm exploits at each run a different randomized version of the original learning sample. In that way the classification method produces an ensemble of more or less strongly diversified models. Then, the predictions of these models are aggregated by a simple average or by a majority vote scheme [GEURTS et al. 2006].

Our hardware implementation exploits as much parallelism as possible. In particular, we realized that the problem become heavily memory bounded when extensively parallelized and this is because there are numerous memory collisions; our aim is to automatically/transparently resolve those collisions with a minimum loss in terms of performance. In particular, one of our aims was to implement an embedded memory subsystem allowing for the efficient execution of the OpenTLD algorithm as well as similar multimedia algorithms involving the Random Forests classifier. Our approach is tailored to reconfigurable devices, as they provide an excellent way to customize the scale and the parameters of the scheme in order to adapt to different applications and cost demands. To support this case we show that our scheme accelerates the underlying algorithm in a "transparent way" allowing the user to change any of the classification parameters seamlessly.

In the following sections we describe in detail the organization of our proposing hardware scheme which is successfully prototyped on a modern embedded FPGA device.

6.2. The OpenTLD hardware architecture

The basic idea behind the proposed approach is that even if the memory access pattern, in the case of Random Forests' classifiers, is random and can be badly distributed over the memory address space we are able to re-distribute it so as to allow for numerous parallel memory accesses. In that way we ensure that we can fully exploit the algorithm's inherent parallelism without the need to use local window caches such as in [BECKER et al. 2011]. The scheme proposed in [BECKER et al. 2011] even though it works for a small number of cores, it will induce serious issues as the number of cores scales since: a) each local cache applied to a single core is fed from the same on-chip image memory or the same fully shared local bus, and b) local caches are increasing the on-chip memory usage in a linear way to the number of cores even in the case that the image is stored off-chip.

The OpenTLD [KALAL et al. 2009] algorithm originally, as the vast majority of the Random Forest based classifiers, has a very inconvenient memory access pattern; almost all the parallel memory queries triggered by the parallel processing cores are referring to the same memory block. As a result, even a distributed memory architecture could not provide the necessary memory bandwidth without significant data repetition. To address this problem we applied a 1-1 scrambling process in the memory addressing that enabled us to reduce this collision rate drastically. Figure 6.2 shows our scrambling scheme which is applied during the memory loading process: Our system scrambles the addresses in such a way that the concurrent memory accesses are allocated to different memory sub-blocks.

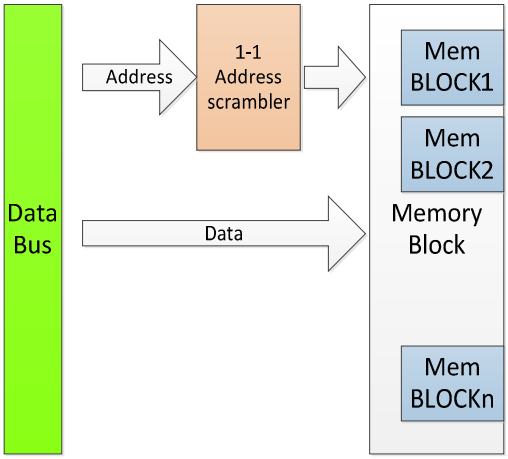

Figure 6.2. Address scrambling process, assuming a Data Bus loading scenario

During the query process the 1-1 address scrambler is applied again in order to re-map the actual addresses into our "scrambled" address space and retrieve the correct data. In order for this scheme to work properly, two issues have to be solved. Firstly and more importantly, the scrambling process should be properly selected so as to maximize the parallel accesses.

In the Random Forest classifiers, we realized that the accesses were performed in a randomized pattern within a sliding window, which had certain characteristics though: The accesses were bounded in the vast majority of the cases (during a complete run) within a distance equal to the size of a memory block (1/32 of the image or 9600 pixels). Keeping this in mind we tried to redistribute the pixels contained in a single memory block to a much larger address space. Therefore, by inversing the order of the address bits, which is a 1-1 function with almost zero latency when applied to hardware, we heavily decreased the memory collisions. This characteristic of the memory access pattern was found in all the Random Forest based applications we have investigated. Secondly, in order for our system to preserve the system's consistency, when more than one parallel queries access the same block (i.e. we have a collision) the queries are serialized. Figure 6.3 demonstrates our approach.

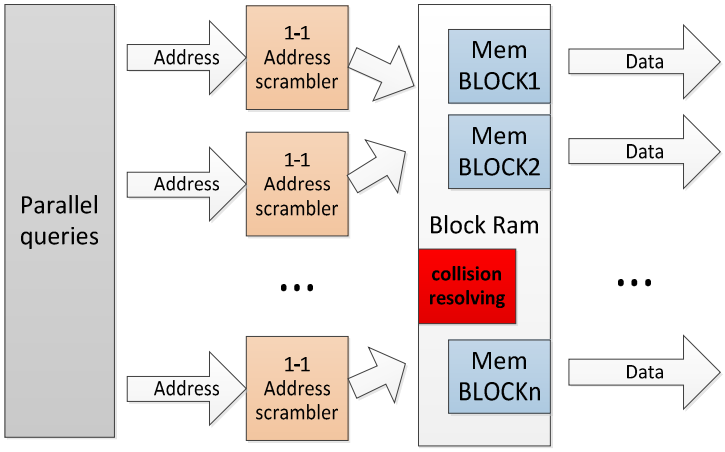

Figure 6.3. Random parallel queries referring on different block rams

Moving to the high-level architecture of our classifier which utilizes our novel scheme it comprises of three basic modules, the Loop Decoding module, the Memory module and the Computation module as shown in Figure 6.4.

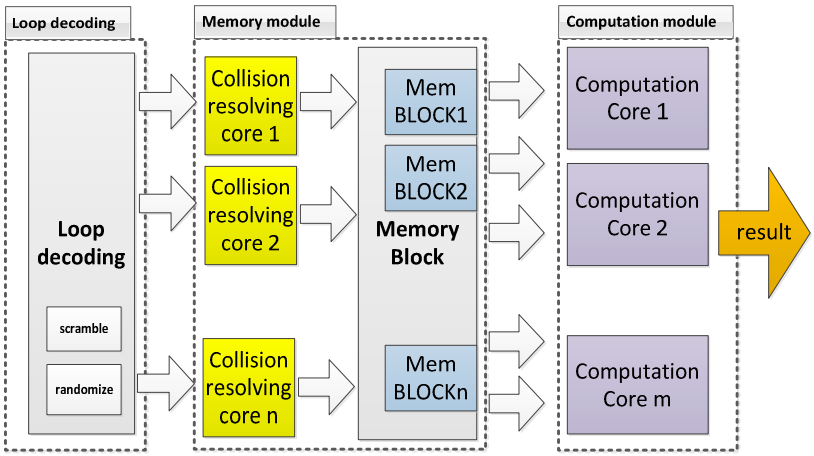

Figure 6.4. High level architecture

The loop decoding module implements the algorithm-specific tasks; in our case study it is responsible for utilizing the sliding window movement, the tree traversing, the scale change while it is also responsible for the randomized sampling if/when applied to a certain window. Its output is a number of addresses corresponding to the different parallel queries. In our case study we produce 16 queries per loop iteration.

The memory module comprises of all the memory blocks that form the classification memory of our system and contain the integral image. Each memory block is connected with a collision resolving core responsible for serializing the memory accesses to this specific block, in case of collisions. We explain that in detail in subsection 6.2.2

The computation module implements the classification function; in our case it just calculates the function A-B-C+D.

6.2.1. Loop decoding module

In this module we actually decode the main loop of openTLD scheme which accesses the random forests. In particular, we implement the sliding window movement function as well as the randomized sampling method which uses the predefined coefficient matrices. Figure 6.5 shows an overview of this module.

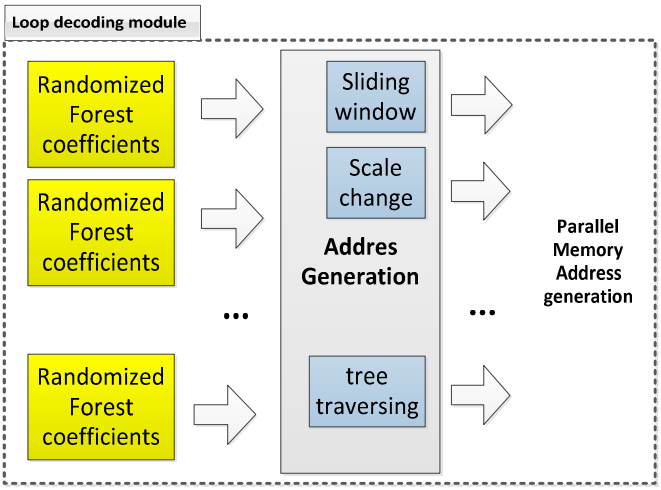

Figure 6.5. Loop decoding module

The loop decoding module produces 16 memory accesses per cycle after decoding 5 continuous iterations of the inner loop of the original openTLD code. It was primarily implemented with an 8-state Finite State Machine combined with 4 lookup tables (i.e. where the randomized coefficients are stored) which are connected with 4 fixed-point multipliers. In order to increase the performance we used 4 hardwired fixed point multipliers provided by the FPGA vendor. These multipliers can execute 1 computation per clock cycle with a tunable pipeline depending on the desirable clock

rate. After investigating the different options, we have selected a 4 stage-pipelined multiplier in order to achieve a moderate 200 MHz clock rate; this is the maximum clock rate that the rest of the modules can support.

6.2.2. The Memory Module

This is the main module of our system and it can be utilized in all the applications that are based upon the Random Forest classifier.

As the memory accesses are random and should be performed in parallel, we have to match the query addresses to the available memory blocks in an optimal manner. In order to achieve that a collision detection module monitors whether there are any colliding memory accesses. The non-colliding accesses are routed to the corresponding memory block while the colliding ones are serialized appropriately.

The serialization module implements a series of very simple finite state machines (in a cascade interconnection) and can serialize a single memory access in every clock cycle. Obviously, the collisions in one block do not affect in any way the accesses in the other blocks; in other words we may end up with an out-of-order execution of the memory accesses but this does not affect the correctness of the Random Forest classifier in any way. Figure 6.7 shows how all the collision detection cores are combined together with the appropriate memory blocks so as to implement our novel memory scheme.

Figure 6.6 demonstrates the dedicated collision resolving module which is attached to each block and which processes all the queries' addresses simultaneously (we draw only 4 queries in this figure for simplicity reasons). The A0,B0,C0,D0 latches are referring to 4 address registers which hold the 4 different memory addresses until the query is completed.

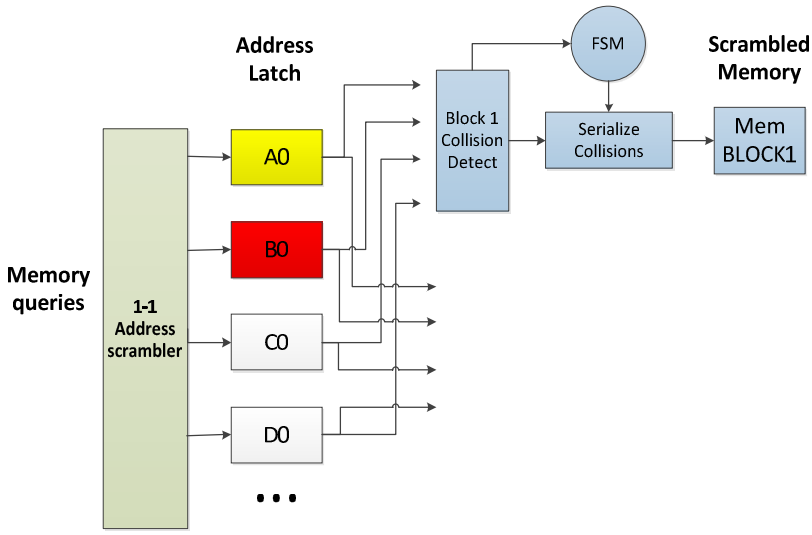

Figure 6.6. Collision detection module organization.

As the input addresses, which are kept in the address registers, can refer to any of the available memory blocks we have to label them (shown with different colors in the figures) in order to track the corresponding data when those are sent from the Memory module to the computation module (the one shown in green in Figure 6.7), since this can be performed out-of-order.

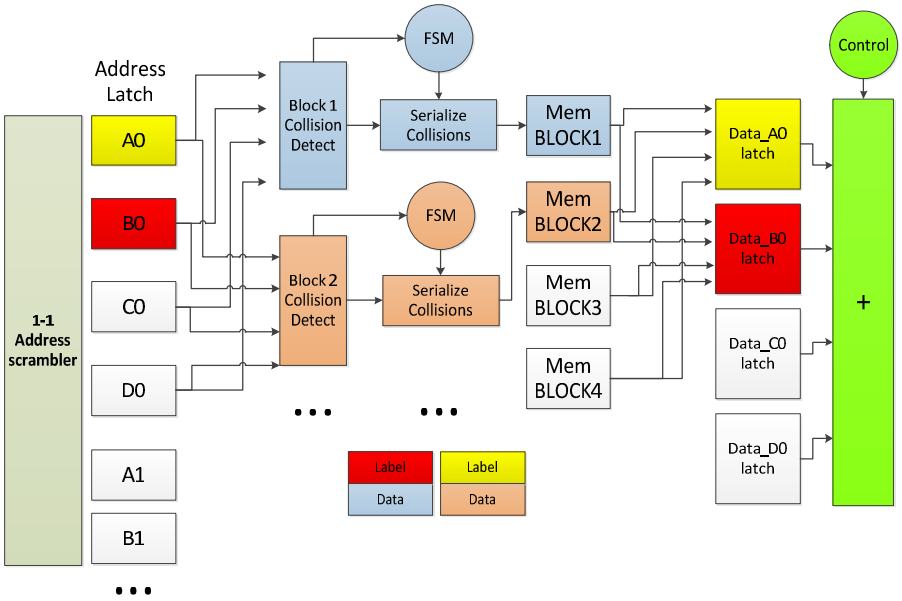

Figure 6.7. Memory module microarchitecture

6.2.3. The Computation module

The computation module is taking the data output of the Memory module and performs the feature computation and the actual classification function; in our case it is a simple sum (i.e. A-B-C+D). It also takes as input the labels corresponding to each data item as the operand's data can be transferred on any of the 32 memory output ports. After the application specific computation is executed the necessary, in every Random Forest classifier, likelihood table memory lookup is performed in order to determine if the examined object is part of the trained dataset or not. Figure 6.8 demonstrates the architecture of this module.

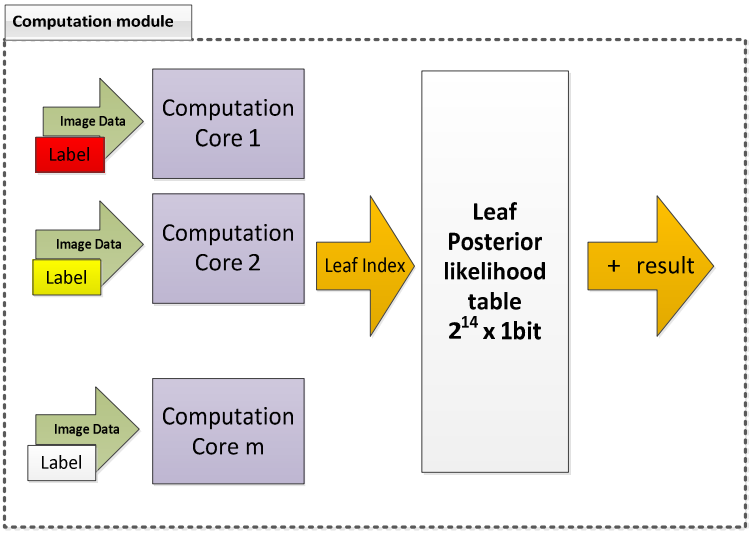

Figure 6.8. Computation Module

The update of the Leaf Posterior likelihood table is a relative easy task since only a small number of entries are updated at the frame-processing stage.

6.3. Distributed Memory

Initially and in order to verify our novel approach we have utilized it in a real-world experiment. In particular, in order to get the memory access pattern to the classification object (i.e. the integral image) we intercepted the algorithm OpenTLD execution in software and recorded its memory behavior. The inputs were real frames from a webcam and initially we have not altered/optimized the algorithm in any way. We decomposed all the nested loops and using the original random function we stored all the memory accesses in testbench files. We then recorded the

memory patterns from our hardware system using the same input and the same values for the random function and we concluded that our memory sub-system handles correctly all the memory accesses.

We also used those real-world memory access patterns for measuring the performance of our system. In particular, we have carefully studied how the sequential memory accesses are distributed to each memory block with and without our scrambling. We assumed 32 equally distributed memory blocks for a 640x480 frame. Each block contains about 10K addresses while the complete frame requires about 300K.

Table 6.1. Collision rate statistical analysis

Memory Distribution in blocks	Average collisions without scrambling[2]	Average collisions[1] 1-1 scrambling	Memory access[3] per cycle (dual port)
8	6.99	2.65	6.06
16	14.98	3.15	10.16
32	24.5	3.6	17.78

After experimenting with several hashing/scrambling functions we realized that the optimal one just inverses the order of the address bits; all the others, like crc16 and crc32, were producing similar performance results whereas they were more complicated in terms of hardware implementation. In particular, reordering the address bits in hardware is a simple and zero latency function. The selected function redistributed the previously neighboring addresses into different blocks with a high probability as proved by our experimental results.

In Table 6.1 we present a statistical analysis over the total memory accesses ($6.81 * 10^6$) needed for the object detection on a single frame. The average collision metric

[2] average collisions per 32 memory accesses
[3] in average after a series of real-wolrd experiments

shows how many queries per loop execution are referring to the same memory block.

Those results clearly demonstrate that, if a conventional memory distribution scheme is used, increasing the number of blocks does not result in any increase in the performance since virtually all the memory accesses target the same block. In other words the classification scheme cannot be parallelized efficiently. After applying our scrambling function we reduced the average collision rate to 3.6 collisions per 32 memory accesses (i.e. by almost an order of magnitude) while the standard deviation is very small (we have up to 4 collisions in the worst case). As a result by using our very simple scrambling scheme and a distributed memory we are able to perform 17.78 memory accesses per cycle (in average) using 32 dual-port on-chip memory blocks.

6.4. The openTLD Device utilization

The system described in Section 6.2 has been initially prototyped in a Xilinx Virtex-6 VLX130T device. The utilization of the device is demonstrated in Table 6.2 and the critical resource was the number of the Memory Blocks, as expected. We selected this device, since in our case study, we could load on-chip a full 640x480 frame, in the integral format which uses 27-bits pixel depth.

Table 6.2. Hardware Cost on a Virtex-6 VLX130T device

Logic Utilization	Used	Utilization
Number of Slice Registers	1736/160000	1%
Number of Slice LUTs	6801/80000	8%
Number of fully used LUT-FF pairs	1307/7230	18%
Number of Block RAM/FIFO	237/264	89%
Number of BUFG/BUFGCTRLs	1/32	3%
Number of DSP48E1s	4/480	0%

Chapter 7

Performance and evaluation results

In this chapter we give the performance and evaluation results of all our novel schemes. We also analytically compare those results with related systems utilizing FPGA devices, multi core CPUs and GPU devices.

7.1. OpenSurf Evaluation and Performance results

In order to measure the efficiency of the OpenSURF system, and since no other similar devices have been presented so far, we compare our performance with (a) that achieved when this algorithm is executed on a high-end Intel processor and (b) the performance achieved when a modified version of the algorithm is executed on a high-end GPU. The source code we used for the evaluation has been downloaded directly from [WEB_SURF] and it is distributed under the GNU General Public License v3.

7.1.1. Accuracy

By altering the arithmetic system utilized as well as the scaling factor of the input image, as it has been described in Section 4.3, and in order to better map it to our reconfigurable device, we have reduced the actual accuracy of the complete detection. In order to provide a quantitative analysis of the introduced error we compared the results produced by our hardware with those reported by the original openSURF software. The errors introduced by our approach altered the total number of common detected points (integer value) and the scale of the reported points (decimal value). After examining the results for numerous 640 x 480 images we have figured out that *in all cases* the number of missing or additional interest points reported by our system was less than 1% of the overall reported points. At the same time the scale and orientation error introduced was always less that 0.01%.

After careful analysis of the results we believe that the differences in the detected interest points is mainly due to the rounding errors when calculating the

determinant based on equation (7) in subsection 4.2.2; those errors lead to an incorrect locating of the maximum-minimum determinants.

7.1.2. Performance comparison with a modern CPU

In order to measure the efficiency of our system we have altered the OpenSURF software so as to use the same arithmetic as our system, and we have measured the actual processing times when this software was executed in a state-of-the-art Intel Core 2 Duo 2.4 GHz processor. The actual processing times for the software implementation were measured using Vtune, a tool provided by Intel for accurate software performance analysis.

The clock frequency of our FPGA system is 200MHz. In order to actually measure the performance we have used six sets of images with different characteristics, and we measured the overall processing time in each case. The results are demonstrated in Tables 7.1 and 7.2. Since the algorithm mainly consists of two distinct steps we report the performance results for each of those steps separately, as well as the overall speedup achieved. The variation in the performance results within the sets is negligible (less than 1% in all cases)

As this table clearly demonstrates, the speedup in all cases is higher than 8x and close to 9x. The average 18msec of processing time per image results in about 56 frames per second which is almost double the 30 fps of today's standard video, but it is still highly demanded by certain machine vision applications (e.g. systems utilized in autonomous cars).

It should also be stressed, that, as it was described in Section 4.1, the non-maximal suppression module searches for the interest points throughout all the scales while the number of those interest points varies according to the image. As a result, in the OpenSURF software implementation the number of the detected points affects the performance of the algorithm as the detection procedure runs after the determinant process has finished. On the other hand, in our implementation since the non-maximal suppression is fully pipelined with the determinant process we achieve a constant performance invariant to the image characteristics.

Table 7.1. Performance results (ms mean values)

Input 640x480	Set one	Set two	Set three	Set four	Set five	Set six
Interest points Hardware (ms)	7,55	7,55	7,55	7,55	7,55	7,55
Interest Points Software (ms)	80.97	63.5	62.47	73,71	69,47	65,4
Orient. Hardware (ms)	9,90	10,25	10,49	11,16	9,19	10,45
Orient. Software (ms)	91,72	90,82	84,8	89,4	100	102,5

Table 7.2. Performance results (speed up mean values)

Input 640x480	Set one	Set two	Set three	Set four	Set five	Set six
Interest Points Speedup	10,7	8,4	8.3	9,8	9.2	8,7
Orient. Speedup	9,3	8,9	8,1	8,0	10,9	9,8
Overall Speedup	9,9	8,7	8,2	8,7	10,1	9,3

7.1.3. Performance comparison with multi-core GPU

In order to further demonstrate the efficiency of our system we compare our performance with that achieved by Terriberry and French [TIMOTHY et al. 2009] who implemented a SURF variation on an nVidia GeForce 880 GTX, GPU. Based on their results, the maximum performance they can achieve is 70fps. Our system can achieve 56fps but it has certain advantages over their approach:

- They use an internal threshold (H0) which reduces the number of detected points and as the detected points affect their performance this accelerates further their approach; this also results in less detected points (i.e. lower accuracy) than our approach

- Their performance drops when the detected points are increased as they follow the standard software architecture which doesn't utilize the determinant and non-maximal suppression pipelining.
- They did not conduct a formal accuracy comparison with the original SURF algorithm or the OpenSURF but they compare their scheme to the predecessor SIFT algorithm
- Their system alone consumes about 200W of power while our device consumes less than 20W. The FPGA itself consume less than 5 watts (based on the Xilinx XPower Estimator (XPE) [UG440]).

7.2. The RFCH Evaluation and Performance results

In this Section we demonstrate the performance of our FPGA-based hardware and we compare it with the optimized single threaded software provided by the inventors of the algorithm when executed on a state-of-the-art low power Intel SU7300 dual-core ULV CPU @ 1.3GHz; such a CPU can typically be used in an embedded multimedia device.

The configuration of the embedded system demonstrated is as follows:

- Number of Calculate Clusters cores: 16/32
- Number of Cluster Features cores: 16/32
- Number of Calculate RFCH cores: 8
- Image Size: 640 x 480
- Number of Clusters (N): 80
- Number of features(f): 7
- Maximum distance (dmax): 4

Those configurations trigger the best performance-to-silicon results while the selected algorithm's parameters trigger the optimal accuracy-to-processing ratio based on [EKVALL et al. 2005]. The modules are clocked at 350 MHz on both FPGAs. Table 7.3 and Table 7.4 show the average time needed for each function in order to process and classify a common object/scene of the CODID dataset [CVAP] in our

FPGA-based system. The performance variance when processing different images is negligible (in terms of processing time for each function invocation). The speedup triggered over the optimized software is about 20-60 times for the first two functions, whereas for CalculateRFCH is only 6.7 times; however this last function takes less than 10% of the total processing time and, as it has been described in Section 4, the main reason for implementing it in the reconfigurable hardware was to minimize the amount of data that should be sent to and from the CPU. It should be noted that in those numbers the I/O overhead has not been taken into account but this has been separated from the critical path due a double buffering scheme we have implemented; this scheme is analytically described in the next paragraph.

Table 7.3. Typical Speedup achieved for processing of a CODID image - 16 cores

Module	SW Time (sec)	HW Time (sec) -	Speedup[4]
Calc.Clust./obj	0.3200	0.0170	18.8
ClusterFeat. /obj	0.5000	0.0175	28.8
ClusterFeat./scene	1.8100	0.0927	19.5
Calc.RFCH/obj	0.0500	0.0074	6.7[2]

[4] Without the I/O time and CPU processing time

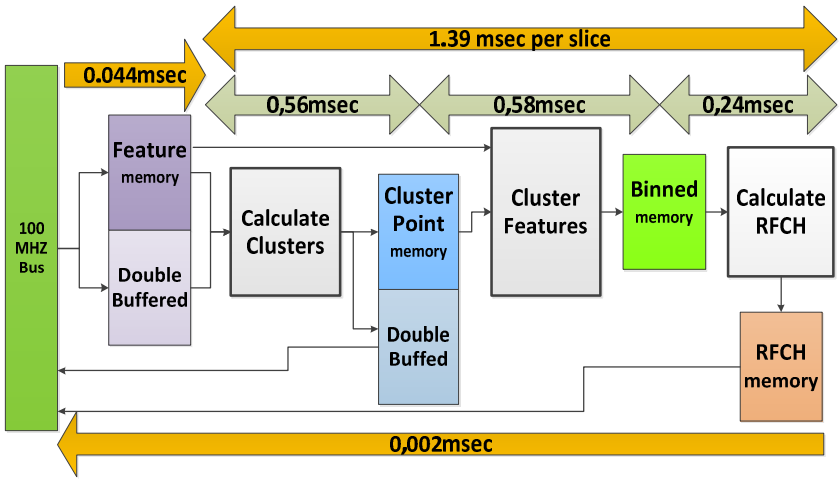

Figure 7.1. Timing Distribution when a 32bit Bus clocked at 100Mhz is used for I/O

Figure 7.1 demonstrates our proposed double buffering scheme which isolates the intercommunication overhead, between the CPU and our hardware modules, from the critical path. In order to measure the communication cost we calculate the transactions needed in order to fully load the Feature memory with 144Kbits, assuming a typical 32bit bus with 4-byte burst mode clocked at 100MHz. In a typical AMBA bus those 4480 transaction needed a total of less than 0.05msec. Moreover, the measured 0.002msec write back time is indeed very low as we need to transfer the RFCH result only 1 time per image (not per image slice).

Table 7.4. Typical Speedup achieved for processing of a CODID image - 32 cores

Module	SW Time (sec)	HW Time (sec)	Speedup[2]
Calc.Clust./obj	0.3200	0.0086	37.6
ClusterFeat./obj	0.5000	0.0088	57.6
ClusterFeat./scene	1.8100	0.0463	39.0
Calc.RFCH /obj	0.0500	0.0074	6.7[5]

[5] We keep the same 8 cores between the two implementation since each such core must be able to process the next 4 lines (i.e. 32 lines in total), based on the fact that we have selected the distance parameter to be equal to 4, and we process in both cases 32 lines at any given time.

The internal hardware is always clocked at 350Mhz. Since our processing time can take, in the very best case, 1.39msec and we use double buffering in the Feature memory (i.e. when we load a certain slice in one part of the Feature memory we simultaneously process the previous slice from the other part), our critical path consists only of the actual processing on our cores.

By trying various dataset configurations with different numbers of objects and different scenes (thus changing the number of times each module processes a certain slice of the image), the performance triggered is listed in Table 7.5.

Table 7.5. Overall Speedup at 350 MHz including the software execution

Configuration	Overall Speedup at 350 MHz (16 cores)	fps	Overall Speedup at 350 MHz (32 cores)	fps
3 objects - 1 scene	11.5	1.1fps	19.65	1.87 fps
10 objects - 1 scene	15.9	1.2fps	27.12	2.04 fps
10 objects - 10 scenes	15.1	1 fps	25.67	1.7 fps

The slight decrease when we go from 1 scene to 10 scenes is due to the fact that the time consumed by the ClusterFeatures and the CalculateRFCH functions is growing while the time consumed by the CalculateCluster stays stable between the two cases. This is because CalculateClusters is executed only during training; in the testing case only the two other functions are executed (several times each one), and as CalculateRFCH has a lower speedup over the others it slightly decreases the overall performance. The average frame per second number demonstrated includes both the detection and the training of the subjects in different scenes. Moreover, it

should be noted that the variance between the different experiments was insignificant.

Those results clearly demonstrate that our system which can trigger a speed of about two fps can be utilized in a state-of-art multimedia system whereas no existing object recognition system running in software can be utilized even at those relatively low object recognition speeds.

7.2.1. multi-core systems performance and energy consumption

In order to investigate how our algorithm behaves on a multi-core, multi-threading environment we have also developed an openMP version of the original algorithm which we have hand-optimized so as to get the best possible parallelism. As clearly demonstrated in [AHMED et al. 2010] and [VIJAYALAKSHMI et al. 2011] the power consumption is increased when more cores and/or more threads are utilized in a multi-core, multi-threaded system. As a result we investigated if the increased speedup triggered by utilizing numerous cores and threads can counterbalance the increased demands in terms of power; if this is the case the total energy needed for the execution of our application can be reduced.

In our experiments we have utilized two Intel multi-cores each with a nominal power consumption of 80 Watts per CPU. "*Talos*" is a dual processor machine hosting a 4-core CPU on each socket (Intel® Xeon® Processor E5620 with 12M Cache, 2.40 GHz, 5.86 GT/s Intel® QPI) while it also supports hypertheading (i.e. it has 8 physical cores while it can support 16 active threads in total). The other machine "*Iraklis*" hosts two CPUs with 4 cores each that do not support hyperthreading (Intel® Xeon® Processor E5430, 12M Cache, 2.66 GHz, 1333 MHz FSB).

In order to investigate how the system behaves when the number of utilized cores and threads increases we executed a number of experiments each exploiting a different number of threads. The speedup is expressed relative to the performance of the single-threaded software; our aim is to demonstrate how efficiently the reference software can be parallelized in terms of both power and performance. As Figure 7.2 clearly demonstrates, the maximum speedup achieved in our experiments is about 5x; this speedup is triggered when we utilized more than 32 threads which

were executed on the 8 available cores. From those results, it is clear that the multi-threaded software is faster than the single-threaded one, since the application very frequently moves data from the memory to the CPU, and the utilization of multiple threads hides this memory latency.

Another interesting conclusion is that *Iraklis* performs much better than *Talos* (as shown in Figure 7.3) although it features an older CPU without hyperthreading while both systems feature the same amount of cache (12MB per CPU, 24MB in total). The difference in the performance comes from the fact that the memory sub-system of *Iraklis* is faster than that of *Talos* due to a faster CPU-memory interconnect system.

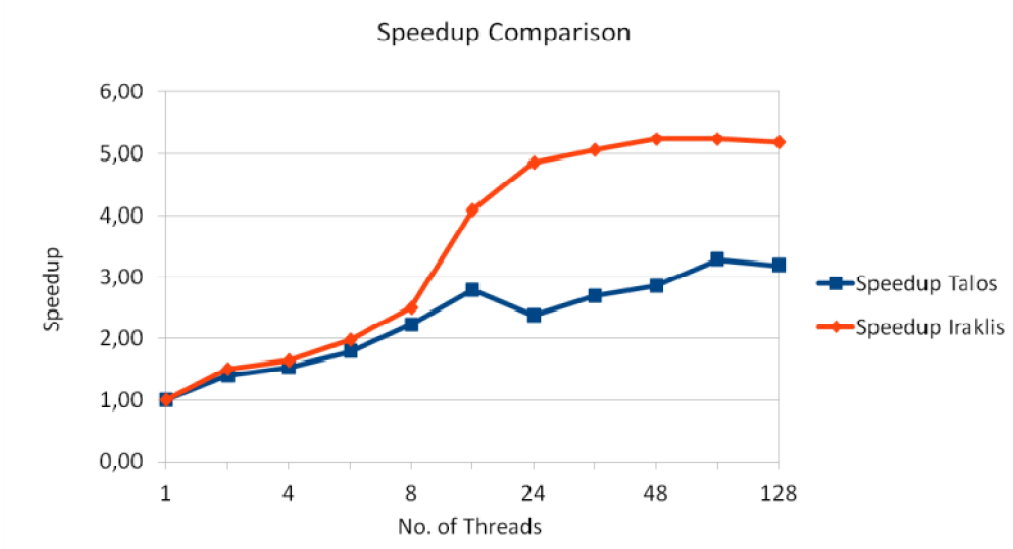

Figure 7.2. Speedup vs Number of Threads for our experiments

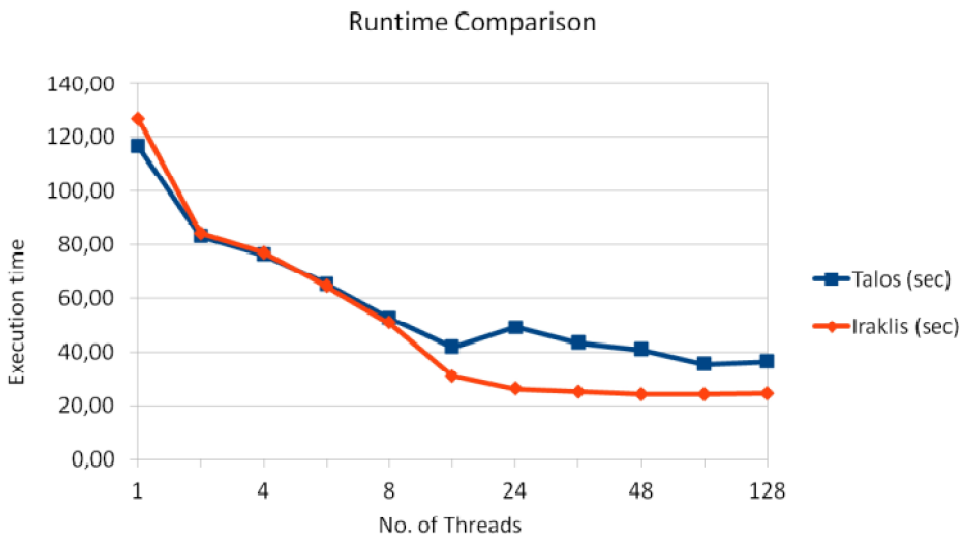

Figure 7.3. Execution time (sec) for 14 CODID images vs Number of Threads

7.2.2. GPU implementation

In this subsection we are going to present the performance results from an implemented version of the RFCH using two modern GPU devices. This GPU version of the RFCH algorithm was also optimized in order to give use a more reliable comparison with the Hardware version presented above. All the performance figures were computed by using very accurate timers written in C and provided by NVidia [WEB_NVIDIA]. We also used the NVidia Visual Profiler, an excellent tool for evaluating the performance of a CUDA application.

Applied Optimization on the GPU modules

In our initial GPU design after we verified its correct execution, we applied the following list of optimizations:

- We utilized Constant Memory.

  In the calculate cluster module, the ClusterPoint[] and the SqrRadius[] arrays, were initially stored in the Global Memory which introduces a considerable latency. Since these arrays are relatively small, we decided to store them in the faster but limited size Constant Memory. Constant Memory is faster but variables stored there have read-only permissions.
- We eliminated unnecessary data transfers to the host.

We tried as much as possible not to transfer unnecessary data from and to the GPUs for a smaller execution time. For example, the CalculateRFCH detecting module needs the binnedImage[] array which is generated in the ClusterFeatures detecting module. We could have copied the result back to the Host and pass it to the CalculateRFCH module by reference and then the later module could copy it back to the Device again. Instead, when the binnedImage[] is created, it just resides in the Global Memory waiting for the CalculateRFCH to utilize it.

- We utilized the *Shared* memory of the device. This is as a very important CUDA optimization as shared memory is a very fast on-chip memory fragment and able to use parallel techniques such as parallel reduction or parallel scan.
- Optimizing block sizes. Based on various tests we performed and trying different block sizes we concluded in an optimal block size for each kernel we utilize.

7.2.3. Overall speedup on the GTX 285

The first GPU set-up we used was based on an GTX 285 GPU. The following table shows the performance results.

Table 7.6. The GTX 285 performance for different object-scene setups.

Configuration	Software time (s)	Hardware time (s)	Speedup
1 object - 1 scene	28.28	4.19	6.7
4 objects – 1 scene	114.30	14.85	7.7
7 objects – 1 scene	202.35	25.24	8.0

7.2.4. Overall speedup on the GTX 580

The first GPU set-up we used, was based on a GTX 580 GPU. The following table shows the performance results.

Table 7.7. Experiments on our system by increasing the objects on the GTX 580.

Configuration	Software time (s)	Hardware time (s)	Speedup
1 object - 1 scene	30.54	3.07	7.89
4 objects – 1 scene	123.3	9.6	12.85
7 objects – 1 scene	218.06	16.67	13.1

As we expected, the GTX 580 managed to accelerate our modules faster than the GTX 285 because of its superior capabilities and resources. The speedup figures though are presented a little bit pushed-up due to the worse performance of the GTX 580 host CPU.

Both GPUs achieved a decent speedup for the ClusterFeatures module especially for the detecting phase as we achieved to off-load all the CPU work; the kernel does almost everything. An average of over 500x speedup have achieved in this particular function (excluding the I/O overhead)

The CalculateRFCH() was a very challenging and interesting implementation for a CUDA port. Its branch divergence, a while-loop inside the body of the algorithm, the unexpected number of bins each thread will produce and the calculation of a histogram were serious challenges for a decent speedup and parallelization. Thus, a fair 8x average speedup is achieved in this particular function (excluding the I/O overhead).

When those results are compared with the FPGA implementation we see that the average GPU speedup according to Tables 7.4, 7.5 is about 10x. The FPGA gave as 20x-25x in average but compared with a slower low power CPU than those hosted on the GPU servers. Thus, we should expect a slightly lower speedup (about 10-15%) when we compare the overall performance achieved with the FPGA supported by a low-power CPU and that one with the GPU supported by a high end server CPU.

7.2.5. Energy Consumption in multicore enviroment

Moving to the power consumption both CPUs consume very close to their nominal power (i.e. 80Watts) when triggering the maximum speedup (i.e. 5x for *Iraklis* and 3x

for *Talos*), whereas the power consumption in all the experiments was more than 60 Watts. As a result the processors consume about 150Watts in total while processing, in the best case, a single CODID image every about 1.7 sec. When comparing those numbers with the corresponding ones achieved by our FPGA system (2.7Watts for 0.5 sec per CODID image) our novel device is about 188 times better in terms of energy consumption.

At the same time the recently introduced 8-core power efficient Xeon CPU (E5-2448), which is optimized for embedded applications [XEON2400], has a nominal power of 70W while it includes 8 cores working at 1.8GHz each. So even if this new 1.8Ghz device has the same performance with our reference 2.66GHz multi-core system, our FPGA-based device will still be 88 times more energy-efficient than this state-of-the-art Intel multi-core.

Moving to another CPU family, the recently introduced ARM-based Cortex-A9, implemented in the newest CMOS technology, has a processing time which is very close to that of the reference Intel CPU; on the other hand this power-optimized ARM consumes about 400mW per core [CORTEXA9]. As a result our novel reconfigurable system is from 10 to 12 times more power efficient than even this state-of-the-art ARM CPU when executing the exact same object recognition scheme. Also looking at the future ARM-based multi-cores, they are expected to be "up to an order of magnitude more power efficient" than the current Intel-based ones [BOLARIA 2009]; as a result, and based on our real-world measurements on the Intel multi-core, our FPGA-based device will still be at least 8 times more energy efficient that those future power-optimized ARM-based multi-cores.

7.2.6. Energy Consumption in the FPGA system

Considering the power consumption of the proposing system, a typical range for the FPGA is 3-4 watts. Thus our system is significantly less power and energy hungry than the conventional software approach.

The specified Intel ULV CPU, in particular, consumes on average about 8W, when executing the specified three tasks on a single core and the other core is disabled, whereas its maximum power consumption is up to 10W[ULV][SU7300]. All the

software power measurements are based on Intel's Power Gadget 2.0 tool. On the other hand, our larger Virtex-6 VLX130T consumes about 2.7W on average (with a peak of less than 3.5W) (based on the Xilinx XPower Estimator (XPE) [UG440]) while it is also from 20 to 25 times faster than the CPU. As a result, the total energy consumed by our FPGA system is 60 to 80 times less than that of the existing single-threaded, purely software based approach.

7.2.7. Energy Consumption in the GPU system

Considering the GPU's power consumption, a typical range for the GPU power consumption would be over 200 watts. Even if the speedup between the FPGA platform and the GPU is at comparable levels, (as mentioned in subsection 7.2.4 the FPGA's performance is compared with a slower ULV CPU than this used for reference for the GPU performance numbers), it is still by far less power hungry[6]. To be more specific, the energy consumption of the FPGA when execute the exact same task is more than 2 orders of magnitude less than the energy consumed by the GPU.

7.3. The OpenTLD Evaluation and Performance Results

7.3.1. Performance Results

In order to evaluate the performance of the proposed scheme we have executed the same application in a state-of-the-art CPU 2.4GHz dual core CPU and on our targeted FPGA which was clocked at a moderate 200MHz clock. The Intel CPU executed the detection process in 80.4 msec in average (the variation is very small), when only one core has been activated.

Then, initially, we executed the complete algorithm on our FPGA prototype utilizing a conventional memory subsystem which accesses a single block at a time; the performance of this approach was virtually the same with the one of the CPU; this was because the classification problem, when utilizing the Random Forest approach, is memory bound, and in our prototype we have used an on-chip memory which has roughly the same bandwidth as the one connected to the Intel CPU. When we increased the number of memory blocks (i.e. created a distributed memory) and

[6] A further study (after the publication of this work [NIKITAKIS et al. 2012]) on the performance results confirmed that the difference between the ULV CPU and the server CPU when running the reference RFCH software in a single thread was no more than 15% at any case.

utilized our novel memory scrambling scheme we ended up with a considerable speedup over the software as well as the conventional approach in hardware. As clearly demonstrated in Figure 7.4. and Table 7.8 our performance grows linearly with the number of blocks and this is due to our very simple, yet very efficient memory scrambling module.

Based on our measurements our system performs the detection task in 1.92msec, in average, for each frame while the variation is negligible. This number does not include any I/O overhead between the FPGA and the CPU which will perform the pre-processing tasks as well as the visualization of the results. In the next section we describe the complete autonomous embedded system and in this case the I/O is also taken into account.

Table 7.8. Performance evaluation on a Virtex-6 VLX130T device at 200Mhz

Memory partitioning in blocks (dual port)	Average collisions (with scrambling)	Speedup @ 200 Mhz vs Single Core CPU	Effective Memory BW (GB/sec)
1	32	0.96	0.29
8	2.65	14.64	3.54
16	3.15	23.27	5.95
32	3.6	41.98	10.42

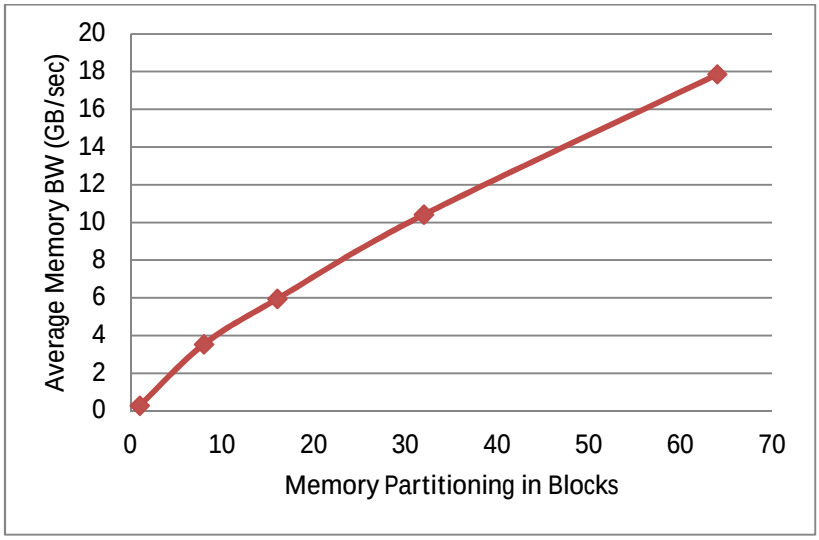

Figure 7.4. Average effective Memory BW vs Number of Memory Blocks[7]

To further prove that the speedup triggered by our novel system is very high even when compared with a modern multi-core CPU utilizing a dual channel memory, we implemented an openMP version of the original algorithm and utilized both cores of the 2.4GHz Intel CPU. The measured speedup, against the two-core implementation, triggered by our FPGA approach when 32 block memories were utilized is 26.16; as a result we demonstrate that even the dual channel memory system of a state-of-the-art CPU featuring 3MB of L3 cache cannot outperform our approach which relies on the 10GB/sec average measured bandwidth that our distributed memory system is providing.

7.4. OpenTLD Embedded design

7.4.1. Communication cost

The recently introduced Zynq-7000 FPGA family from Xilinx allows for the implementation of real-world embedded systems exploiting both the performance of an FPGA accelerator as well as the flexibility of a modern dual core low power ARM CPU in the same die.

In the implementation of the complete embedded system (the Virtex-6 implementation of the last sub-section covers only the actual core of the OpenTLD

[7] We have implemented and tested up-to 32 dual port memory blocks

which is the detection task and not the pre-processing and the visualization of the results) we utilized the very low-cost Avnet's Zedboard [WEB_ZED] which is powered by a Xilinx Zynq-7000 SoC (XC7Z020).

Since we could not fit a complete 640x480 integral image in the memory blocks of this low-cost device, we have used a smaller image size (480x360) in order to measure the real world performance and then we projected the measurements to the 640x480 frame size initially used.

Our test platform was set around the 12.04 Ubuntu linux distribution for ARM, loaded with the latest OpenCV library. The connection between the dual-core ARM and the reconfigurable resources has been realized through the Xillybus IP core [WEB XILLY]. Xillybus offers several end-to-end stream pipes for data movements between the host CPU and the hardware resources at very high data rates. As all the stream pipes are buffered using dedicated FIFOs (shown in Figure 7.5) we are able to separate the two different clock domains of the bus and our hardware modules.

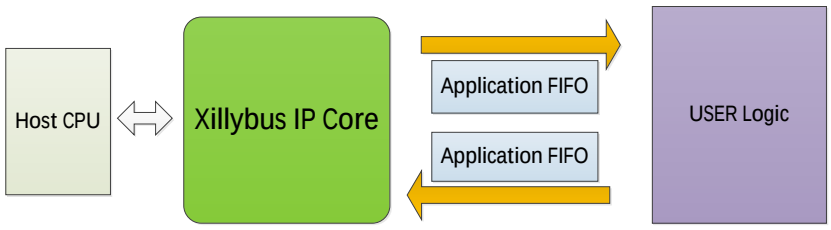

Figure 7.5. Xillybus IP core communicates data with the user logic through a standard FIFO

For our experiments we set the bus clock at 100MHz and our hardware scheme at 200MHz. The Xillybus core supports higher clock rates at the bus, but due to some limitations on the DMA controller for the specific ARM processor the use of a higher clock rates in the bus proved to be useless.

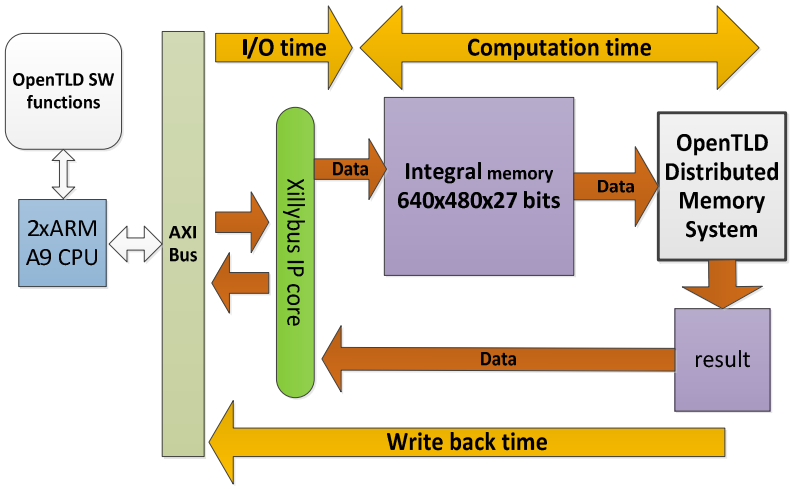

Figure 7.6. Xilinx Zynq-7000 verification scheme

In our proposed architecture, demonstrated in Figure 7.6 we transfer data from the CPU to the hardware modules practically only when we load the Integral image memory; then our hardware modules process those data until the complete image is fully processed.

In order to minimize the communication overhead in this embedded application we chose to suppress the amount of data we had to move to the hardware side. This was possible by generating the integral image in hardware and thus instead of moving 640 x 480x27 bits (i.e. size of integral image) from the CPU to the FPGA fabric we had to move only 640x480x8 bits (i.e. size of the original image); this triggers an almost 4 times reduction.

It is known [VIOLA et al. 2001] that the integral image can be computed efficiently in a single pass given a grayscale image i(x,y) using the formula:

$$I(x,y) = i(x,y) + I(x-1,y) + I(x,y-1) - I(x-1, y-1),$$

where I(x,y) is the Integral image and i(x,y) is the grayscale image

As Figure 7.7 shows it is possible to compute on the fly the integral image data as we transfer the pixel data to the hardware modules by caching only 2 integral image lines (the current and the previous ones). After each integral image pixel is

computed it is redistributed to the memory blocks using our novel address scrambling scheme described in Section 5. The aforementioned approach is easy to be synchronized with the streaming data coming to the system as the Xillybus core fully decouples the different clock domains providing synchronous read/writes from the Host to hardware parts and vice versa.

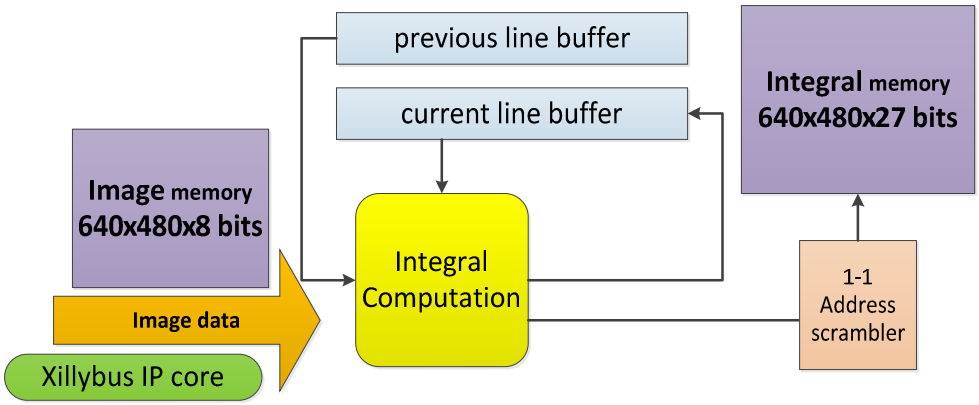

Figure 7.7. Integral computation scheme

When our system handled a 480x360 image we measured the bus bandwidth to be 200Mb/sec. This is much lower than the 370Mb/sec official bandwidth reported by Xillybus, when working at 100MHz, and it was due to a certain limitation of the DMA controller. Based on those measurements we need 1.46msec for the loading of an 640x480 image on the current configuration which will be lowered to 0.79msec when the problem with the DMA controller is addressed. The hardware processing time is 1.92msec in average as mentioned in subsection 7.3.1, so if we use a double buffering scheme (i.e. using smaller images or utilizing a larger device) we can hide the communication latency completely. Even if no such scheme is utilized and we have to add the intercommunication overhead to the hardware processing overhead our system will still be more than 15x faster than a dual core Intel CPU.

The intercommunication time is dominated only the transfer of the image since the result which should be sent back to the CPU is a single 32bits datum and this is only needed once for each complete image. The update of the Leaf posterior likelihood array does not also trigger any significant overhead in the communication as only a

few hundred values have to be updated between frames and this can be done fully in parallel with the actual processing.

Even though we may have a performance decrease because of the communication overhead (i.e. if the on-chip memory size does not allow for double buffering) our autonomous embedded system has still very high performance which can be translated to either more fps or to support the very important today multiple object tracking. In this latter configuration the hardware processing time will be increased while the communication time will stay stable and thus we will minimize further the performance loss due to the data transfers.

7.4.2. OpenTLD GPU implementation

In this sub-section we describe the implementation of the detection part of the OpenTLD algorithm in a highly parallel GPU platform in order to have a comparison with our FPGA proposing scheme. In this experiment we also have a Host CPU which executes the complete algorithm except of the detect() function which is executed on the GPU device. The high level architecture of the GPU implementation is depicted in Figure 7.8. The GPU utilized in the NVidia GTX 285 device and it is programmed using the well established CUDA API [WEB_NVIDIA].

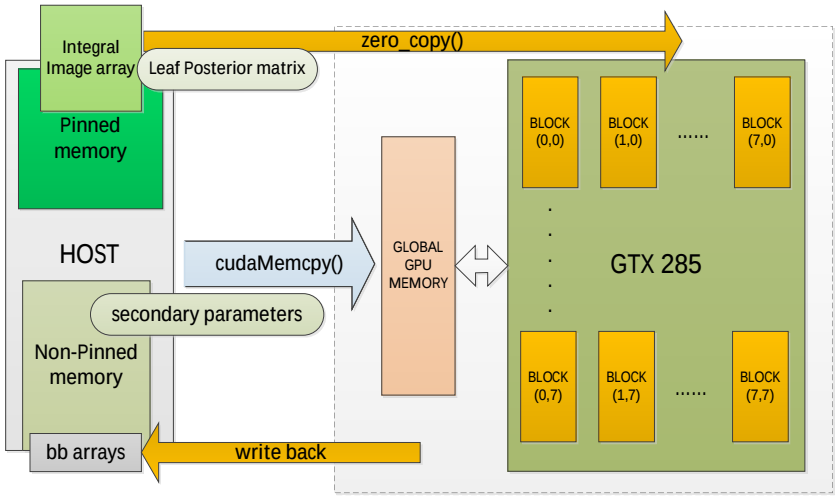

Figure 7.8. High level architecture of the Detector Module

The kernel module we built in order to perform the detection phase needs basically the Integral image (640x480x32 bits) array and the Leaf Posterior matrix (2^{14} bits)

which are computed in the Host exactly as in our embedded system. After the completion of the processing of each frame the host has to update these two arrays.

Since the CPU has a considerable amount of work for copying the above arrays to the GPU's memory, we decided to use one of the asynchronous memory copy APIs. Those CUDA APIs won't block the CPU thread as the default API would. In this manner the host stream will keep processing the data without stalling, and the result is an obvious reduction in the triggered I/O overhead.

For the Integral image array we used the zero copy memory optimization of CUDA. Zero copy enables GPU threads to directly access the Host memory and forces the allocation of a page-locked host memory that the CUDA runtime can track.

All asynchronous memory copies as well as the zero copy method require a mapped pinned memory which is used for arrays that reside at the Host side of the module. Pinned memory is allocated using the cudaHostAlloc() function, and prevents the data of being swapped out. So once pinned, that amount of memory becomes unavailable to other Host processes and the Device can fetch it without any intervention from the CPU using DMA (direct memory access).

In the CUDA API we set the dimensions of the computational grid, as follows:

- 64 two-dimensional blocks and
- 16 two-dimensional threads in each block

The above setup activates a total number of 1024 kernels (i.e. threads); 1024 is the maximum number of iterations we might need to parallelize depending on the width and height of the integral Image and it proved to trigger the highest possible performance on the specified GPU. Those kernels mainly calculate the sum over the Integral image: A-B-C+D as mentioned in Section 5 while utilizing the Leaf Posterior matrix in order to compute a probability value. This value is used to determine the bounding box which contains the object which is under tracking.

By testing several video samples and setting training objects of different shapes and sizes, we confirmed a considerable acceleration to all those experiments when compared with the single-core CPU performance. In Table 7.9 we see the

performance of the reference single core CPU, the GPU and our embedded system considering the detect() function.

Table 7.9. Performance Evaluation in terms of speedup

Accelerated Entity	Software time (msec)	CUDA time (msec)	Speedup vs single-core	Embedded[8] time (msec)	Speedup vs single-core
Detect() with I/O	80.4	7.5	10.6x	3.38	23.78x
Detect() no I/O	80.4	3.05	26.36x	1.92	41x

As those results clearly demonstrate our embedded system outperforms the GPU by at least a 2x factor when the I/O overhead is triggered while this speedup will be much higher when the DMA controller problem is addressed (3x speedup) or a double buffering scheme is utilized (4x speedup). Even without taking the I/O improvement into account our embedded device can perform the actual processing at a higher rate than a modern GPU.

Moving to the energy consumption the CPU has a nominal power consumption of 14W when one core is utilized and the GPU consumes more than 85W, while our system can consume at most 4W. As a result the proposed embedded device consumes at least 40x less energy than either the CPU or the GPU when executing the OpenTLD complete applications

7.5. OpenTLD existing hardware schemes

Moving to the comparisons with the existing hardware approaches, to the best of our knowledge, there is none that has implemented the Forest Trees structures in hardware in such a generic, not application-specific, way. The only system that partially implements such a classifier is the one in [BECKER et al. 2011]. However, in order to parallelize the system they have used a different approach based on caching the image data for the parallel classifiers which takes advantage of specific

characteristics of the application, poses many restrictions and thus it is certainly not applicable to other applications utilizing Forest Trees.

Even when looking at the actual performance triggered by the application-specific approach of [BECKER et al. 2011], it is not possible to have a direct comparison in terms of speedup with our system as they have not implemented a publicly announced OpenTLD clone but instead their own proprietary one. In general, though, our system triggers a speed-up of more than 40 in comparison with a single core implementation whereas they claim a speedup of just 5 when using their own software as a reference running on a much slower CPU. More importantly, our approach has certain advantages when compared with the one in [BECKER et al. 2011]:

— Our system can support any combination of forest trees and classification features without changing a single wire in our hardware scheme. We have implemented up to 15 trees while supporting more than 12 features and our performance remained constant as the statistical characteristics of memory accesses hasn't been affected. In [BECKER et al. 2011] they can only support 10 trees and 10 features due to hardware restrictions.

— Our system architecture is not setting any restriction in the sub-window size. In [BECKER et al. 2011] the sub-window size is set to 1024 pixels.

— In their system the more processing cores they utilized the more memory blocks they need, as they are used as local cache. In our case the number of processing cores can be increased without any need to increase the number of memory blocks. We just split the existing memory in more slices and get a sub-linear bandwidth increase[9] as shown in Figure 7.4.

[9] under the assumption that the clock speed remains constant

Chapter 8

Computer Vision Embedded design

Based on our experience from the development of the previously mentioned systems, a software/hardware co-design strategy for Computer Vision application offer the opportunity to accelerate numerous algorithms using a common generic architecture. Based on our approach the camera interface modules will stay untouched, while we will change the software running on the CPU and load different bit streams in the FPGA using the run-time reconfiguration aspects of the devices so as to accelerate different algorithms.

In this chapter we analyse our approach and combine the knowledge of the previous case studies in order to propose a unified field-customizable architecture for computer vision (CV) algorithms.

8.1. A unified field-customizable architecture for CV algorithms

In such a unified architecture we need an efficient way to communicate through various buses and among different FPGA devices. This functionality is being currently provided by the Xillybus IP core [WEB_XILLY] and we are fully exploiting its potential through this thesis.

Furthermore in the proposing architecture a user is able to reprogram the FPGA device by altering the bitstreams of various accelerators depending on the application needs. By using the Xillybus IP core the design is being "communication transparent" as the user only handles a numbers of FIFOs and control registers. All the rest of the bus signaling is being handled by the Xillibys IP. The Xillybus IP core is also portable among various Xilinx devices utilizing the AXI bus (e.g Zynq devices) as well as AXI over PCI (e.g Virtex 6 and Virtex 7 devices). The rest of the Host's environment is built around a Linux operating system (such as Ubuntu 12.4) which also provides extended flexibility too. In our case studies we mainly used the Zedboard development board [WEB_ZED] as a host platform. Zeadboard is a very

low cost board although it is able to support a full Linux distribution running on its low power dual ARM A9 CPU.

Based on the proposed approach the user customizes the software-side accelerator driver to communicate efficiently with the underlying HW accelerator cores on the FPGA side as well as with any software libraries. If the interface is the same for all the HW cores a single platform will be able to support different algorithms, customized on–field, with a simple bitstream download. This architecture is depicted on Figure 8.1. In this thesis we have implemented and tested the RFCH HW accelerator (covered in Chapter 5) as well as the OpenTLD HW accelerator (covered in Chapter 6) utilizing the Xillinx Zedboard development board and following the aforementioned design strategy.

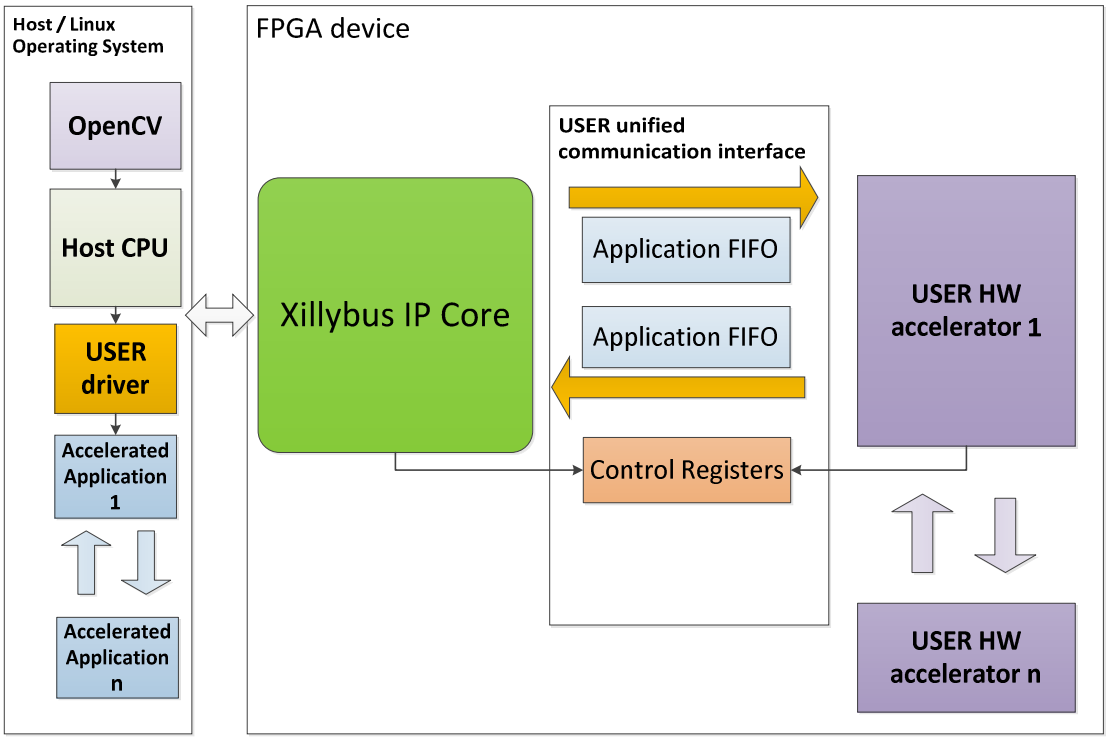

Figure 8.1. A unified embedded architecture for computer vision

Our unified embedded system is able to reprogram itself on demand and change its functionality from an object detection accelerator (aka RFCH) to a long term tracker (aka OpenTLD) without any modification, just by loading the new off-the-shelf software, with our unified HW drivers (See below) and the appropriate bitstream file.

8.1.1. User driver

The user driver proposed is based on a very simple, yet effective communication with the low-level Xillybus kernel host driver utilizing the Linux /dev files; the goal is to support the transfers to and from the accelerator in a transparent way.

The kernel host driver actually generates device files which behave like named pipes: They are opened, read from and written to just like any file, but behave much like pipes between processes or TCP/IP streams [WEB_XILLY]. The Xillybus kernel driver provides to the user a separate /dev file (e.g. /dev/xillybus_fifo_32) for each FIFO or control register used for the intercommunication. Using only these building blocks a user is able to develop a custom driver providing all the necessary data transfer and control functions for the correct and efficient operation of the HW accelerator. The following snippet of code shows an example of such functionality:

```
……
memory_fd_r = open("/dev/xillybus_mem_8", O_RDONLY);
 if (fd_r < 0) {
   if (errno == ENODEV)
    fprintf(stderr, "(Maybe a write-only file?)\n");
   perror("Failed to open devfile");
   exit(1);
……
}
int read_mem_8(int pos) {
unsigned char data;
   if (lseek(memory_fd_r, pos, SEEK_SET) < 0) {
   perror("Failed to seek");
   exit(1);
   }
   allread(memory_fd_r, &data, 1);
return (int)data;
}
```

Furthermore the same driver supports any Xillybus IP core configuration: The streams and their attributes are auto-detected by the driver as it's loaded into the host's operating system, and device files are created accordingly.

Also when the driver is loaded, the necessary DMA buffers are allocated in the host's memory space, and the FPGA is informed about their addresses automatically. The number of DMA buffers and their size are separate parameter for each stream. These parameters are hardcoded in the FPGA IP core for a given configuration, and are retrieved by the host during the discovery process. [WEB_XILLY]

8.1.2. Xillybus core

The Xillybus IP core consists of an FPGA IP core and a low level driver for the host: All the low-level design considering the bus initialization and signaling is hided from the user as it is automatically supported by the core itself [WEB_XILLY]. Moreover, Xillybus actually supply a wrapper for the underlying transport (e.g. a AXI/PCIe DMA engine), and offers several end-to-end stream pipes for application data transport.

As shown in Figure 8.1, the Xillybus IP core transfers data through a standard FIFO (Application FIFO), which is supplied by the IP core's user. This gives the FPGA designer the freedom to decide the FIFO's depth and its exact interface with the internal acceleration module.

It is also very importat that the the Xillybus IP is platform independent (supports both Xillinx's and Altera's devices) so a single design can be implemented in different FPGAs, based on the end-user requirements.

8.1.3. Non-Accelerated Code

The non-accelerated code is actually the part of the application that is not receiving any acceleration by the HW and is completely necessary as it makes all the initialization and control work in order to transfer the required data to the accelerator. It contains the original algorithm's functionality while it may have some minor modifications so as to fully utilize the underlying HW (for example it may use a different numerical system than the original scheme). The original piece of SW is augmented so as to make the HW calls to the accelerator by using the user driver

described in the previous subsection. The following snipped of code shows and example of how those HW calls are performed.

```
void CalculateClusters_hw (unsigned char **feature,  unsigned char
**clusterPoint, unsigned long *sqrRadius)

{

  int i, f;

  int numFeatures=7;

  int numBins= 80;

  for (i = 0; i < numBins; i++)

    for (f = 0; f < numFeatures; f++)

                    clusterPoint[f][i] = unsigned
char)(255*(float)rand()/RAND_MAX);

  hw_ClusterPointArrayToBram(clusterPoint);

  hw_ChipReset(4); //resets all

  hw_ChipEnable(1); //Enable CalculateClusters

……
```

In the above snippet the CalculateClusters_hw() function actually replaces the original CalculateClusters() while providing the exact same functionality as the original piece of software code.

8.1.4. User HW accelerator

The user hardware accelerator is the actual implemented FPGA core which is synthesized for the specific target platform; In our case the FPGA device which powers the Zedboard is the Xilinx Zynq-7000 SOC (XC7Z020-CLG484-1) [DS190]. The HW accelerator is connected to the interface provided by the Xillybus core as shown in Chapter 6 for the case of OpenTLD algorithm. The RFCH scheme presented in Chapter 5 is also designed with the same reference platform in mind. In the above fashion an embedded systems designer is able to build a library of HW accelerators supporting numerous different computer vision algorithms while they will all be

fully compatible to the reference embedded architecture. As a result the end embedded systems can be implemented in different FPGA-based systems; for example, and based on the interportability of the Xillybus sub-system, the same SW/HW combination can be executed on a Xilinx Zynq board containing an on-chip dual-core ARM or an Altera board which hosts an external Intel CPU.

Chapter 9

The OpenTLD framework Generalization

In this chapter we attempt to generalize the OpenTLD proposed architecture in almost any algorithm that utilizes the Viola and Jones framework. We revisit the OpeSURF case [BAY et al. 2006] presented in Chapter 4 and going further with the popular CascadeClassifier [WEB_OPENCV] retrieved from the OpenCV library. We compare our new findings with the already presented case of the OpenTLD [KALAL et al. 2009] from Chapter 5.

9.1. Problem formulation

As shown in the subsection 2.4 the Viola and Jones problem is a main approach in a very large number of implementations. It may has the form of the originally presented Haar-like features extraction from the Integral image along with a cascade classifier, or a more general form of feature extraction (i.e box filters) from the same Image memory. The presence of a classifier or not, does not change the problem's characteristics as we are going to show in the OpenSURF revisited case [BAY et al. 2006]. Figure 9.1 shows how the use of the Integral Image, transforms the computation of any rectangle filter into only 4 memory accesses and 3 additions.

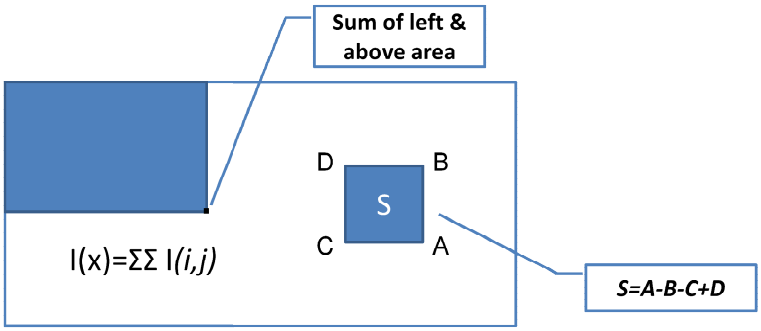

Figure 9.1. The Integral image concept

This is the key property that makes all similar algorithms to demand the exact same operations in different data (i.e the Integral image data). As a result algorithms based on the Viola-Jones framework formulate a very well defined memory bound problem.

According the aforementioned considerations we pose the Viola and Jones problem like the SIMD (Single Instruction Multiple Data) concept found in Computer Architecture literature. In other words we have to apply the same bunch of operations (aka box filter computations) in a large amount of independent data (data level parallelism). The Figure 9.2 depicts the above concept. A very large amount of memory accesses applied in the same Integral image memory, to form the operands of the same computation (i.e A-B-C+D).

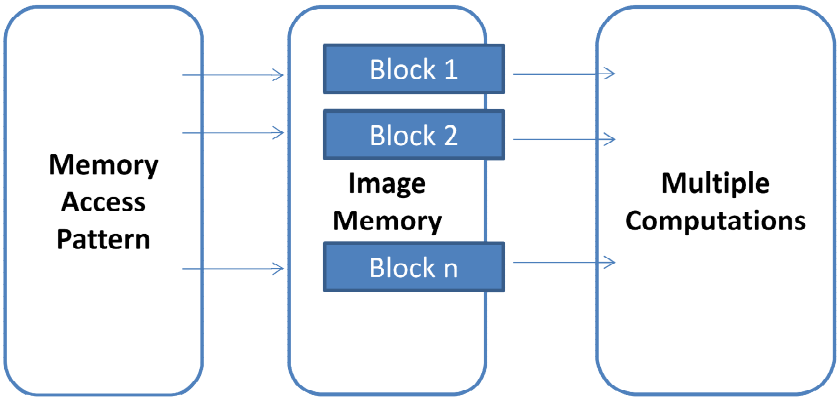

Figure 9.2. A generalized Viola-Jones framework problem formulation

As shown in the next subsections, all of our experiments show a consistent behavior in the memory characteristics (i.e the memory access pattern).

9.2. A Generalized Viola and Jones Hardware Architecture

Having the aforementioned concept in mind we organize our proposing embedded architecture as in the OpenTLD case: It consists of three independent modules connected in a pipeline fashion. The Loop Decode Module, the Memory Module and the Computations Module.

Such architecture has a very important property: It decouples the memory sub-system organization from the application-dependent memory access pattern. Thus if we could provide a high and constant performance for this memory subsystem at any case we will actually manage to accelerate a whole family of algorithms. Figure 9.3 depicts the overview of the proposed architecture.

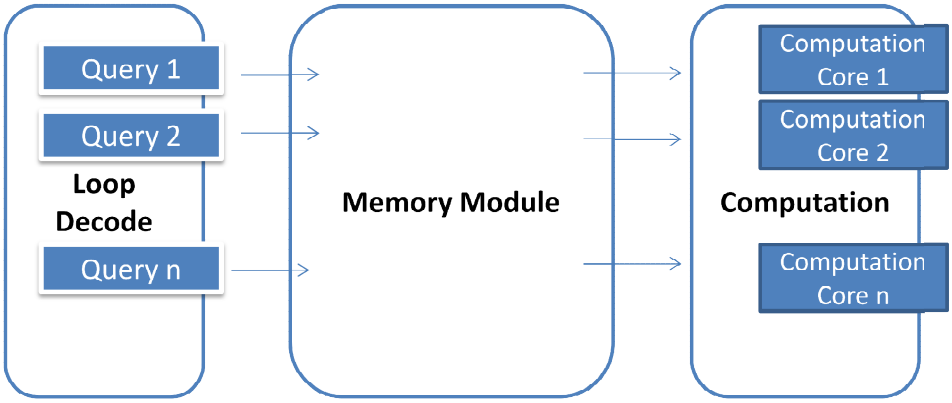

Figure 9.3. An overview of a generalized Viola and Jones system architecture

Furthermore, the algorithm dependent part of an embedded vision system that uses the Viola and Jones framework is much easier to be implemented (even with High Level Synthesis) when our proposed architecture is followed. This comes from the fact that the parallelization is performed automatically from our proposed memory subsystem and it is not a hand-custom work which is heavily dependent on the special characteristics of the underlying algorithm.

In the following subsections we are not going to describe in detail the organization of our proposed scheme as it follows exactly the OpenTLD organization. We are going though to give more details considering the methodology we followed for our experiments and also the performance results that our memory system gives.

9.3. Distributed Memory Analysis

Our novel memory approach is developed and verified in a series of case studies involving real-world experiments using as input real frames from a webcam or various saved images. For this reason we have developed a special tool which simulates at a functional level our proposed memory architecture. It intercepts the memory access pattern of the examined algorithm and performs certain statistics in real-time. It actually monitors the accesses in 32-query windows at the lowest level of granularity of each examined algorithm. It then assigns those queries to memory blocks (i.e 32) and computes the number of collisions of each block. We assumed 32 equally distributed memory blocks for a 640x480 frame. Each block contains about 10K addresses while the complete frame contains about 300K.

We then build a distribution of the maximum collisions occurred in a 32 address query. The maximum collisions number it is actually the collision rate of the worst performing block. We are interested only in the worst performing block as this defines the overall latency of every 32-query bunch. At each complete run of the algorithm we also compute an average collision rate based on those worst-performing blocks which actually expresses the average latency of the memory module on a real data processing. By using these metrics we are able to compute with great accuracy the average throughput of the memory subsystem at any examined case.

All the examined algorithms showed a remarkably consistent behavior regarding the collision rate independently to the processed data (less than .1% variation at any case).

9.4. Memory Subsystem Performance Results

In this subsection we show how the OpenTLD memory subsystem behaves in terms of performance in another two examined algorithms. The OpenSURF and the CascadeClassifier [WEB_OPENCV]. The following table sums-up the performance results for all the three different algorithms.

Table 9.1. Performance Evaluation of our generalized memory subsystem

Algorithm	Blocks (dual port)	Collision Rate	Mem access/cycle	Throughput GB/sec
OpenTLD	32	3.6	17	10
OpenSURF	32	5.5	11.7	6.8
CascadeClassifier	32	6.5	9,9	5.8

From the above results we clearly see that the OpenSurf takes a significant boost from our proposed memory scheme and performs about 12 mem acceses/cycle. The hand-custom parallel implementation we've presented in Chapter 4 performed 8 mem acceses/cycles. It's easy for someone to conclude how much more effective the hardware design will be, and also effortless, if the parallelization of the bottleneck of the underlying algorithm is done automatically from our novel memory

scheme. Furthermore we see that a generic and very well established algorithm the CascadeClassifier from the OpenCV also receives a significant boost from our scheme.

Chapter 10

Conclusions

In this thesis it is presented the first known (at the date of publication [BOURIS et al. 2010]) FPGA-based implementation of one of the most efficient feature detector scheme. This detector, called SURF, is very robust in terms of image rotation and translation, as well as in terms of changes in illumination and scale deformations. The FPGA implementation of the feature detector calculates the location scale and orientation of the interest points at a rate of 56 frames per second. In order to create an efficient system, special care was taken so as to balance the numerical precision supported with the utilization and speed of the hardware resources. The proposed system utilizes a combination of fixed-point and floating-point arithmetic so as to produce highly accurate results at high rates.

The presented FPGA device, although clocked at only 200MHz, is more than 8 times faster than a state-of-the-art CPU executing the same algorithm (and in average about 9 times). Moreover, our device achieves comparable performance with that of a high-end GPU consisting of 128 floating point CPUs clocked at 1.35GHz Moreover, the GPU consumes more than 200W, the dual core Intel 40W (while being 8 times slower) while our systems consumes less than 20W; thus it is in general about one order of magnitude more energy efficient than both a CPU and a GPU.

The SURF case shows that a pure hardware implementation is able to give a significant speedup over an Intel CPU and it is less energy hungry than both a CPU and a GPU. It is hard though to achieve the scalability, flexibility and customization that offered by a software/hardware co-design approach as the RFCH embedded system which is also part of this thesis.

Our RFCH implementation comprises of a complete low-power embedded object recognition system that can support multi frames per second speeds. The presented system can work in a stand-alone manner while it can be considered a general object detection scheme implemented on a reconfigurable device that can execute a very efficient such algorithm at a rate of more than one frame per second while

consuming about 60 times less energy than a low-power CPU executing the exact same algorithm. To be more specific the FPGA gave a 20x-25x speedup in average when compared with a low power Intel ULV CPU; at the same time the average GPU speedup is about 10x when compared to the server's CPU hosting the GPU. Even if the performance between the GPU and the FPGA is at comparable levels, when power is taken into account the FPGA is at least two orders of magnitude less energy hungry.

Our novel system can efficiently be utilized in multimedia systems or wearable computing systems performing complex object recognition (such as fixed and mobile game consoles or tomorrow's smartphones).

We clearly demonstrate that such a complex task can, probably for the first time (at the date of publication [NIKITAKIS et al. 2012]), be addressed by a single chip solution running on minimal power; this is achieved by exploiting the heterogeneity of custom hardware and a low power embedded CPU. We present such a single chip prototype while our ideal target single-chip platform is the recently introduced Xilinx Zynq-7000 single chip device featuring a dual-core ARM CPU and FPGA reconfigurable logic in the same silicon [DS190].

The RFCH system revealed that if special attention is given to communication between the software and the hardware accelerator, complex vision schemes are able to be accelerated with less development effort (than a pure hardware solution) while maintained the software flexibility.

Furthermore, our third scheme (the OpenTLD) pushed further the above arguments and showed that even when the algorithm itself has non-trivial memory accesses that cannot be handled by classical memory interleaving schemes, it is still possible to extract significant speedup targeting only the bottleneck of the memory subsystem.

We thus proposed and implemented a simple, yet effective distributed memory sub-system, upon which we efficiently parallelize and implement, as an autonomous

embedded system, the OpenTLD tracking scheme; furthermore our distributed memory sub-system is independent of the various software parameters.

Our real-world measurements demonstrate that the speedup achieved by our embedded system over a modern CPU is more than 20x while our device is even faster than a highly parallel GPU. Moreover, our system consumes more than 40x less energy than the CPU and the GPU. Since our system is also very flexible, modular and low-cost, it can be efficiently utilized in numerous multimedia and robotic applications which involve the Random Forest approach.

The proposed architecture, when prototyped on a state-of-the-art FPGA which incorporates a dual core ARM CPU, can execute this high-end detection algorithm at a rate of more than 20 times higher than that triggered when a modern dual-core CPU executes the exact same detection scheme; this speedup comes mainly from the utilization of our novel memory sub-system. At the same time our embedded system is faster even from a high-end GPU, while it consumes two orders of magnitude less energy than the conventional CPUs and GPUs. Moreover, the proposed memory scheme is modular, flexible, easily expandable and application-independent.

The OpenTLD case reveals more interesting findings. Algorithms that are using a similar to the Viola and Jones framework are able to be accelerated from the same memory sub-system. The simulation of both the OpenSURF algorithm and the CascadeClassifier from the OpenCV library shows promising results and worth further exploration.

All the above proposed hardware architectures independently show the various design strategies that a embedded systems designer has to follow in order to achieve high performance and low energy consumption when utilizing FPGA devices in embedded/hardware vision systems.

Finally, in order to support the above arguments we propose a unified reconfigurable hardware architecture which is able to support different computer vision algorithms while also being customized on-field. This can be achieved through

the exploration of the heterogeneity of the custom hardware when coupled with a low power embedded CPU. As a proof of concept we implemented both RFCH as well as OpenTLD in a very low cost development board the “Zedboard” [WEB_ZED]. Zedboard is a single chip prototype powered by the recreantly announced Xilinx Zynq-7000 single chip device featuring a dual-core ARM CPU and FPGA reconfigurable logic in the same silicon.

References

AHMED YOUSSEF, MOHAMED ZAHRAN, MOHAB ANIS, AND MOHAMED ELMASRY, " On the Power Management of Simultaneous Multithreading Processors", in IEEE Transaction on Very Large Scale (VLSI) systems, vol 18, no 8, August 2010

BAY H., TUYTELAARS T., AND VAN GOOL L.. "Surf: Speeded up robust features". European Conference on Computer Vision, 1:404-417, Graz, Austria, May 2006

BENEDETTI, PERONA P., "Real-Time 2-D Feature Detection on a Reconfigurable Computer,", pp.586, 1998 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'98), 1998

BECKER T., LIU Q., LUK W., NEBEHAY G., AND PFLUGFELDER R.. 2011. Hardware-accelerated object tracking. In Proc. Int. Conf. on Field Programmable Logic and Applications (FPL), Sept. 2011.

BISSACCO A. AND GHIASI S.. "Fast visual feature selection and tracking in a hybrid reconfigurable architecture". In 2nd Workshop on Applications of Computer Vision, ECCV 2006, Gratz, May 2006

BOLARIA JAG, "Intel, ARM Battle for Microservers", Processor Watch, May 17, 2012

BOUGANIS C.-S., CHEUNG P. Y. K., NG J., AND BHARATH A. A.. "A steerable complex wavelet construction and its implementation on FPGA". In 14th International Conference on Field Programmable Logic and Applications, volume 3203 LNCS, pages 394-403, January 2004.

BOURIS, D.; NIKITAKIS, A.; PAPAEFSTATHIOU, I., "Fast and Efficient FPGA-Based Feature Detection Employing the SURF Algorithm," *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on* , vol., no., pp.3,10, 2-4 May 2010

BPTLD.2011. https://github.com/Ninjakannon/BPTLD.git

BREIMAN LEO. 2001. Random Forests. In International Journal of Machine Learning ,Volume 45 Issue 1

BROWN M. AND LOWE D.G.. "Invariant features from interest point groups". British Machine Vision Conference, Cardiff, Wales, pages 656-665, March 2002.

BROUSSEAU, B.; ROSE, J., "An energy-efficient, fast FPGA hardware architecture for OpenCV-Compatible object detection," *Field-Programmable Technology (FPT), 2012 International Conference on* , vol., no., pp.166,173, 10-12 Dec. 2012

CHANGJIAN GAO AND SHIH-LIEN LU, "Novel FPGA based HAAR classier face detection algorithm acceleration", International Conference on Field Programmable Logic and Applications, pp.373 - 378, 2008.

CHE M. AND CHANG Y., 2010. A Hardware/Software Co-design of a Face Detection Algorithm Based on FPGA. *In Proceedings of the 2010 International Conference on Measuring Technology and Mechatronics Automation* - Volume 01 (ICMTMA '10), Vol. 1. IEEE Computer Society, Washington, DC, USA, 109-112.

CHO J., MIRZAEI S., OBERG J., KASTNER R., 2009. Fpga-based face detection system using Haar classifiers. In *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays* (FPGA '09). ACM, New York, NY, USA, 103-112.

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

CORTEXA9. http://www.arm.com/products/processors/cortex-a/cortex-a9.php

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.

DS768. Xilinx® , DS768AXI Interconnect

DS190. Xilinx® , DS190, Zynq-7000 Extensible Processing Platform Overview

DUC MINH PHAM; AZIZ, S.M., "FPGA architecture for object extraction in Wireless Multimedia Sensor Network," *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2011 Seventh International Conference on* , vol., no., pp.294,299, 6-9 Dec. 2011

EKVALL S., D.KRAGIC, "Receptive Field Coocuurrence Histograms for Object Detection", in IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 84 - 89, 2005

FIORE P., KOTTKE D., AND GAMPAGNA D. "Efficient feature tracking with application to camera motion estimation". In Conference Record of the 32nd Asilomar Conference on Signals, Systems & Computers, volume 2, pages 949-953, November 1998.

GEURTS PIERRE, ERNST DAMIEN, WEHENKEL LOUIS. 2006. Extremely randomized trees. In Machine Learning 2006, 63(1), 3-42.

GENTSOS CHRISTOS, SOTIROPOULOU CALLIOPE-LOUISA, NIKOLAIDIS SPIRIDON AND VASSILIADIS NIKOLAOS, "Real-Time Canny Edge Detection Parallel Implementation for FPGAs", International Conference on Electronics, Circuits, and Systems (ICECS), pp.499, 2010.

GOSHORN DEBORAH, JUNGUK CHO, RYAN KASTNER, SHAHNAM MIRZAEI, "Field Programmable Gate Array Implementation of Parts-Based Object Detection for Real Time Video Applications", FPL2010

GUMSTIX. http://www.gumstix.com/

HARRIS C., AND STEPHENS M.. 1998. Proceedings of the 4th Alvey Vision Conference, page 147--151. (1988)

HADJITHEOPHANOUS, S.;TTOFIS, C.; GEORGHIADES, A.S.; THEOCHARIDES, T. ,"Towards hardware stereoscopic 3D reconstruction a real-time FPGA computation of the disparity map", in IEEE International Conference on Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1743 – 1748, 2010.

HIROMOTO M., SUGANO H., AND MIYAMOTO R., 2009. Partially parallel architecture for AdaBoost-based detection with Haar-like features. *IEEE Trans. Cir. and Sys. for Video Technol.* 19, 1 (January 2009), 41-52.

KALAL ZDENEK, MATAS JIRI, MIKOLAJCZYK KRYSTIAN. 2009. Online learning of robust object detectors during unstable tracking. In 3rd On-line Learning for Computer Vision Workshop, Kyoto, Japan, IEEE CS.

KYRKOU C.,THEOCHARIDES T., "A Flexible Parallel Hardware Architecture for Ada Boost-Based Real-Time Object Detection," to appear in IEEE Transactions on Very Large Scale Integration (VLSI) Systems (Published online, no.99, pp.1-14, 2010).

KYRKOU C., TTOFIS C., AND THEOCHARIDES T., 2013. "A Hardware Architecture for Real-Time Object Detection Using Depth and Edge Information", ACM Transactions on Embedded Computing Systems, accepted to appear.

LAIKA, A.; PAUL, J.; CLAUS, C.; STECHELE, W.; AUF, A.E.S.; MAEHLE, E., "FPGA-based real-time moving object detection for walking robots," *Safety Security and Rescue Robotics (SSRR), 2010 IEEE International Workshop on* , vol., no., pp.1,8, 26-30 July 2010

LOWE D. G.. "Object recognition from local scale-invariant features". In Proceedings of the International Conference on Computer Vision ICCV, pages 1150-1157, Corfu, Greece, 1999.

LUCAS BRUCE D. KANADE AND TAKEO. 1981. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2* (IJCAI'81), Vol. 2. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 674-679

MAREE RAPHAEL, GEURTS PIERRE, PIATER JUSTUS, AND WEHENKEL LOUIS . 2004. A generic approach for Image Classification based on Decision Tree Ensembles and Local Sub-windows. In 6th Asian Conference on Computer Vision, Volume 2, page 860-865

MAREE, R., GEURTS, P., PIATER, J., WEHENKEL, L. 2005. Random subwindows for robust image classification. In Computer Vision and Pattern Recognition. (CVPR 2005), pp 34 - 40 vol. 1

MCCREADY R., 2000. Real-Time Face Detection on a Configurable Hardware System. *In Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications* (FPL '00). Springer-Verlag, London, UK, 157-162.

MS-9A35.MSI WindBox III (MS-9A35) Core2Duo Fanless Embedded System

MACQUEEN J. B., "Some Methods for classication 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.

NIKITAKIS A.,PAPAIOANNOU S. AND PAPAEFSTATHIOU I. .2013. "A novel low-power embedded object recognition system working at multi-frames per second", ACM Trans. Embed. Comput. Syst. 12, 1s, Article 33 (March 2013), 20 pages.

OSMAN, H.E. 2009. Random forest-LNS architecture and vision. In Industrial Informatics (INDIN 2009). 7th IEEE International Conference on , vol., no., pp.319-324, 23-26 June.

PCI_BUS. Datasheet PCI Bus Bridge Memory Controller, 100 MHZ

SHOTTON J., A. FITZGIBBON, M. COOK, T. SHARP, M. FINOCCHIO, R.MOORE, A. KIPMAN, AND A. BLAKE. "Real-Time Human Pose Recognition in Parts from a Single Depth Image. In proceedings of the Computer Vision and Pattern Recognition conference (CVPR), Colorado Springs CO, IEEE, June 2011.

SU7300.http://ark.intel.com/products/42791/Intel-Core2-Duo-Processor-SU7300-(3M-Cache-1_30-GHz-800-MHz-FSB)

SUMALATHA B., M.SEETHA, G. NARAYANAMMA. 2011. An Efficient Approach for Robust Image Classification Based on Extremely Randomized Decision Trees. In International Journal of Computer Science and Information Technologies, Vol. 2 (2), 677-685

SE S., BARFOOT T., AND JASIOBEDZKI P.. "Visual motion estimation and terrain modeling for planetary rovers". In 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space, Munich, 2005.

TOMASI C. ,KANADE T. "Detection and tracking of point features" Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.

TIMOTHY B. TERRIBERRY, LINDLEY M. FRENCH, AND JOHN HELMSEN, "GPU Accelerating Speeded-Up Robust Features", International Journal of Parallel Programming ISSN 0885-7458 (Print) 1573-7640 (Online) December, 2009.

ULV. http://ultrabooknews.com/2012/02/07/intel-core-ulv-vs-lv/

UG534. Xilinx® , UG534 ML605 Hardware User Guide

UG081. Xilinx® , UG081 Microblaze processor reference guide

UG440. Xilinx®, UG440 XPower Estimator User Guide

VIOLA PAUL AND JONES MICHAEL. 2001 "Rapid object detection using a boosted cascade of simple features", Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on, Vol. 1 (15 April 2003), pp. I-511-I-518 vol.1.

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.

VIJAYALAKSHMI SARAVANAN, SENTHIL KUMAR CHANDRAN,SASIKUMAR PUNNEKKAT, D. P. KOTHARI , "A Study on Factors Influencing Power Consumption in Multithreaded and Multicore CPUs", WSEAS Transactions on Computers, Issue 3, Volume 10, March 2011

VTUNE. Intel® VTune. Amplier XE, Intel® VTune, Amplier XE Documentation.

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.

WEB_FEATURE. http://en.wikipedia.org/wiki/Feature_detection_(computer_vision)

WEB_GLASS. http://www.google.com/glass/start/

WEB_MATCH. http://en.wikipedia.org/wiki/Match_moving

WEB_NVIDIA. http://www.nvidia.com/object/cuda_home_new.html

WEB_RECOGNITION. http://en.wikipedia.org/wiki/Outline_of_object_recognition

WEB_OPENCV. http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html

WEB_SCALES http://www.nada.kth.se/~tony/cern-review/cern-html/

WEB_SURF, Open source SURF feature extraction library, http://code.google.com/p/opensurf1/

WEB_XILLY. http://www.xillybus.com/

WEB_ZED. http://www.zedboard.org/

WITKIN, A. P. "Scale-space filtering", Proc. 8th Int. Joint Conf. Art. Intell., Karlsruhe, Germany,1019--1022, 1983.

XBOX. http://www.xbox.com/kinect

XEON2400.http://www.intel.de/content/dam/www/public/us/en/documents/marketing-briefs/xeon-e5-2400-for-intelligent-systems-brief.pdf