

Pipelined Object Tracking on XMOS Multicore Environment

A Thesis

Submitted by

**MURALI KRISHNA NAIK KORRA
(EE09B055)**

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY

in

Microelectronics and VLSI Design



**DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS**

May 2014

THESIS CERTIFICATE

This is to certify that the thesis titled **Pipelined Object Tracking on XMOS multicore environment**, submitted by **MURALI KRISHNA NAIK KORRA**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology and Master of Technology** in Microelectronics and VLSI design, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Sridharan K

Guide

Professor

Dept. of Electrical Engineering

IIT-Madras, 600 036

Dr.Sudha Natarajan

External Guide

XMOS Semiconductor Pvt. Ltd

IITM

Research park, 600 115

Place: Chennai

Date: 5th May 2014

ACKNOWLEDGEMENTS

I am thankful to **Dr. Sudha Natarajan** for her guidance throughout the project. She understood the work I am capable of and provided me with right directions. She gave me freedom and trusted me to try various things. She helped me a lot in directing me to right materials and research papers whenever I was stuck with doubts. I am also thankful to **Dr. K. Sridharan** for his valuable discussions that helped me stay focussed in my project.

Thanks to XMOS pvt ltd for providing me required boards for my project. I am also grateful for the support provided by the Electrical Engineering department staff for providing resources required during the course of my stay. And thanks to IITM for providing me the best campus and lab environment.

ABSTRACT

KEYWORDS: Object Tracking, Parallel System, Multi-core processor, Frame subtraction, connected components, XMOS.

In this thesis a new system is developed for tracking an object in a video frame in a muticore multithreading environment. Memory storage, object tracking algorithm and displaying the output in Liquid Crystal Display(LCD) all run in different threads simultaneously. The developed system will serve as a test platform for testing any other object tracking algorithm.

The object tracking system uses 6 threads that run in parallel while 1 thread is used initially to load the images into SDRAM in RGB565 format. We have achieved a frame rate of 8 frames/s for an image of size 480x272. In addition, we also did experiments on memory and timing analysis on a multithreaded platform.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
ABBREVIATIONS	viii
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Object tracking	2
1.3 Literature survey	4
1.4 Multicore XMOS architecture	5
1.5 Contributions of the project	6
1.6 Organization of the report	6
2. PIPELINED OBJECT TRACKING	8
2.1 Object tracking	8
2.1.1 Background subtraction	8
2.1.2 Frame subtraction	10
2.1.3 Histogram of Oriented Gradients (HOG)	13
2.2 Issues in object tracking	15
2.3 Connected Component Analysis (CCA)	15
2.4 Single pass algorithm for CCA	17
2.4.1 Neighborhood context	17
2.4.2 Label selection	18
2.4.3 Merger control	18
2.4.4 Data table	19
2.4.5 Table sizes	19

3. IMPLEMENTATION ON MULTICORE XMOS ARCHITECTURE	22
3.1 SliceKIT and setup	22
3.2 LCD Slice	24
3.3 SDRAM Slice	27
3.4 xTIMEcomposer Studio	29
3.5 Thread Scheduling	29
3.6 Implementation	30
3.6.1 Image format	30
3.6.2 Pipelined object tracking system	31
3.7 Results	34
4. MEMORY AND TIMING ANALYSIS	41
4.1 Memory analysis	41
4.2 Space Complexity (Processor memory usage)	42
4.3 Image Processing Operations - I/O and Memory	42
4.4 Resource utilization	44
4.5 Timing Results	45
5. PERFORMANCE IMPROVEMENT	46
5.1 Parallel CCA	46
6. CONCLUSION AND FUTURE WORK	50
6.1 Conclusion	50
6.2 Future work	50

LIST OF TABLES

Table 3.1: Buffer array as stored in sdram	29
Table 3.2: Thread Scheduling	30
Table 4.1: Memory required for different datatypes	41
Table 4.2: Space complexity	42
Table 4.3: Timing results	45

LIST OF FIGURES

Figure 2.1: Background Subtraction	9
Figure 2.2: Flow chart of Background subtraction	9
Figure 2.3: Flow chart of Frame subtraction	10
Figure 2.4: Frame subtraction and binarisation	11
Figure 2.5: Extracting HOG feature	13
Figure 2.6: Histogram binning and interpolation	13
Figure 2.7: Connected components	16
Figure 2.8: Basic architecture of Single pass algorithm	17
Figure 2.9: A label is assigned to the current pixel based on already processed neighbors	17
Figure 2.10: Worst case number of objects	19
Figure 3.1: Slice kit setup	23
Figure 3.2: Slice kit port map	24
Figure 3.3: LCD server	25
Figure 3.4: Processing speed versus number of parallel threads	27
Figure 3.5: SDRAM server	28
Figure 3.6: Image format	31
Figure 3.7: Pipelined object tracking	32
Figure 3.8: Thread diagram	33
Figure 3.9.1: Bounding box has been put over the moving object in frame 1	34
Figure 3.9.2: Bounding box has been put over the moving object in frame 2	35
Figure 3.9.3: Bounding box has been put over the moving object in frame 3	35
Figure 3.9.4: Bounding box has been put over the moving object in frame 4	36

Figure 3.9.5: Bounding box has been put over the moving object in frame 5	36
Figure 3.9.6: Bounding box has been put over the moving object in frame 6	37
Figure 3.9.7: Bounding box has been put over the moving object in frame 7	37
Figure 3.9.8: Bounding box has been put over the moving object in frame 8	38
Figure 3.9.9: Bounding box has been put over the moving object in frame 9	38
Figure 3.9.10: Bounding box has been put over the moving object in frame 10	39
Figure 3.9.11: Bounding box has been put over the moving object in frame 11	39
Figure 4.1: Resource utilization	44
Figure 5.1: Block diagram of proposed parallel CCA	47
Figure 5.2: Passing features case I	48
Figure 5.3: Passing features case II	49

ABBREVIATIONS

APP	Application
CCA	Connected Component Analysis
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
LCD	Liquid Crystal Display
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
SDRAM	Synchronous Dynamic Random Access Memory
TGA	Truevision Graphics Adapter
ZIF	Zero Insertion Force

CHAPTER 1

INTRODUCTION

1.1 Motivation

Pipelining is a natural concept in everyday life, e.g. on an assembly line. Consider the assembly of a car: assume that certain steps in the assembly line are to install the engine, install the hood, and install the wheels (in that order, with arbitrary interstitial steps). A car on the assembly line can have only one of the three steps done at once. After the car has its engine installed, it moves on to having its hood installed, leaving the engine installation facilities available for the next car. The first car then moves on to wheel installation, the second car to hood installation and the third car begins to have its engine installed. If engine installation takes 20 minutes, hood installation takes 5 minutes, and wheel installation takes 10 minutes, then finishing all three cars when only one car can be assembled at once would take 105 minutes. On the other hand, using the assembly line, the total time to complete all three is 75 minutes. At this point, additional cars will come off the assembly line at 20 minute interval.

There's an everlasting pursuit of realism in computational sciences. Traditional serial computing (single processor) has limits in terms of the following:

- Physical size of transistors
- Memory size and speed
- Instruction level parallelism is limited
- Power usage, heat problem

One solution to these problems is **parallel computing**: simultaneous use of multiple processing units to solve one computational problem. The advantages are:

- Saving time
- Solving larger problems
- Access to more memory

- Better memory performance
- Providing concurrency
- Saving cost

Mobile and portable platforms increasingly require the ability to handle images and videos smoothly. Image data is large in size and mobile devices can afford to have only a chip or two to perform the processing. It is therefore important to study computing devices that support parallel processing so that applications run in real-time. Some of the contemporary computing solutions are based on Field Programmable Gate Arrays (FPGAs) and processors that support multi-threading. This project examines the power of the latter. In particular, the project examines the use of multi-threaded and multicore processors from XMOS for performing a typical image processing task namely object tracking, at high speed. We briefly review the basics of object tracking as well as multithreading and then proceed to state the precise contributions of this project.

1.2 Object tracking

Computer vision has become an important application of smart embedded systems used in a wide range of fields ranging from human computer interaction to robotics. Object tracking is a fundamental component of computer vision that can be very beneficial in applications such as unmanned vehicles, surveillance, automated traffic control, biomedical image analysis and intelligent robots, to name a few. Object tracking is used for identifying the trajectory of moving objects in video frame sequences. Like most computer vision tasks, object tracking involves intensive computation in order to extract the desired information from high-volume video data. In addition, the real time processing requirements of different computer vision applications stress the need for high performance object tracking implementations.

In addition, real time performance is another constraint which mainly depends on the algorithm used for tracking. Hence, object tracking is a challenging task. In this report, we deal with moving object tracking problem. All objects with significant motion are tracked.

The most common approach to track objects is to first detect them using background subtraction and then, establish correspondence from frame to frame to find the tracks of the objects. Despite its popularity, background subtraction based methods still lack the robustness to handle specific events such as tracking multiple interacting objects with heavy occlusion, the most unwanted events that often happen in video. In video sequences, these interactions result in several challenges to the tracking algorithm. Since blob generation of moving objects is based on connected component analysis, closed objects generate a single merged object, and in this situation, visual features of the occluded objects are not observed and occluded objects cannot be tracked.

In cases where background is not static we need to follow a completely different approach. An example for non-static background is tracking a car in a highway. Here we need to describe the object based on certain sharp features. These features should be such that, we can differentiate object with that of the background. It is sufficient if we are able to “Detect” the object in every frame. In order to detect the moving car, you can choose color, size, model etc. But the chosen feature can easily match with another car that is passing closer to the car which needs to be tracked. So much sharper features are necessary while tracking cars in a highway. In order to detect clouds, we can take features such as color (blue) and image derivative to get the boundary between cloud and the sky. So with these two simple features, we can detect clouds. But In order to detect a snake, we need to have even more features as snakes are of different colors and size. The features chosen for snake can easily be confused with worms if we missed any feature that distinguishes worms and snakes. So “Object Representation” or “Feature Selection” is very important for detecting an object. A set of features or “Object Descriptor” is similar to DNA or finger print which helps differentiate one object from the other.

From the discussion above, we can see that each object needs special set of features or “Object Descriptor” in order to detect them. Histogram of Oriented Gradients (HOG) is one such object descriptor used mainly for human detection. With added features to HOG, it is possible for human identification but it is much different algorithm than detection. HOG based tracker will be explained later in the report.

1.3 Literature survey

Even though implementing parallel object tracking application is the main objective of the project, the project involved thorough learning about the basic image processing concepts and sequential object tracking for moving on to parallel coding. The textbook by Gonzalez and Eddins [9] provides all the concepts required for the basics of image processing.

Numerous algorithms for object tracking have been proposed. It is a complex task which comprises two main subtasks: i) object detection and ii) tracking. Object detection algorithms can be classified according to Yilmaz et al [3] into point-detection schemes, background subtraction techniques, and supervised learning techniques. Furthermore the tracking portion of object tracking can be performed either separately or jointly with object detection. Tracking aims to generate the trajectory of objects across video frames and Yilmaz et al. [3] characterized tracking algorithms across three main categories: i) point tracking, ii) kernel tracking, and iii) silhouette tracking. In this work we leverage an efficient algorithm which is based on frame subtraction detection. The main focus of this work is on the performance improvement achieved over a software implementation from a carefully-crafted hardware implementation on the Xilinx Slicekit.

Several object tracking algorithms have been implemented on reconfigurable devices in previous works. Nevertheless, one of the biggest challenges of custom hardware implementations is mapping complex algorithms onto reconfigurable fabric architectures that can offer good performance under rigid resource constraints. Usman et al. [4] adopted an FPGA software processor based design to implement mean shift [3] based object tracking. However, the biggest size of tracked objects is limited to 32 x 32 pixels and the performance is very low. For the evaluation of our implementation, we use a Xilinx Slice kit device. We achieve performance of 8 fps for 480 x 272 pixels per frame.

There is an extensive literature on object detection, but here we mention a seminal paper on human detection [7]. In contrast [7] detector uses a simpler architecture with a single detection window, but appears to give significantly higher performance on pedestrian images. Jablonski

and Gorgon [6] have implemented this classic two-pass algorithm on an FPGA. This uses stream-based processing, with a small local window in the first pass, and a point operation (the merge table lookup) in the second pass, giving a latency of one frame.

Several high-speed parallel algorithms exist for connected components labeling [5]. While such algorithms give considerable speed improvement over the classic algorithm, this speedup is achieved through massive parallelism. Such parallelism is very resource intensive, requiring a large number of essentially identical processors. Parallel connected components labeling [2] is also proposed in this report.

It uses very simple processing, but requires multiple passes through the image to completely label an image. Being iterative, it requires a frame buffer to store the intermediate image between passes. The number of passes required depends on the complexity of the region shapes, making such an algorithm unsuited for real-time processing.

A modification [1] to the classic two-pass algorithm enables a single pass implementation, eliminating the need for a frame buffer, and significantly reducing the latency. A single pass algorithm must extract the features of interest for each component while determining the connectivity. This removes the need for producing a labeled image and avoids the second, relabeling pass. However, it requires merging and relabeling on the fly to ensure that consistent results are obtained.

1.4 Multicore XMOS architecture

Real time operation can be accomplished via parallel processing or multithreading. A thread is part of a program which can run independently. Any process or an application program can be split into independent smaller codes which can be made to run as a thread. Multithreading CPUs have hardware support to run each thread concurrently. It can be compared with multiprocessing. The main difference between multithreading and multiprocessing is that, error in one thread can

bring down all the threads in a process whereas an error in one process cannot bring down another process.

For instance, if video compression and pipelined object tracking are two different processes, failure in video compression may not affect object tracking process and vice versa. In object tracking process, we have memory interface, algorithmic unit and LCD display for the output as independent units of the code. Independent here refers to the ability to fetch next image data and store it in memory while processing the current image. However, failure in storing the next image data in memory by a memory interface thread can affect other subsequent threads in the object tracking process.

1.5 Contributions of the project

The contributions of the project are as follows.

1. Design of a pipelined object tracking system that runs on several threads.
2. Interfacing SDRAM and LCD for storage of image sequences to facilitate streaming and processing using different threads and to display the video with results of object tracking.
3. Implementation of object tracking (based on frame subtraction) on an XMOS multi-threaded microcontroller to design an embedded system.
4. Analysis of memory and time for embedded implementation.

1.6 Organization of the report

The remainder of this report has been organized as follows.

Chapter 2 explains the Pipelined processing for Object tracking, Frame subtraction and Connected Component Analysis (CCA) in detail. It also explains issues in object tracking.

Chapter 3 gives an introduction to experimental setup. It explains the parallel processor we used in this project and also gives an idea of how the setup has been developed. Experimental results are provided here.

Chapter 4 gives the memory and timing analysis. It also gives code development and resources consumed.

Chapter 5 gives an approach to improve performance by parallel CCA in detail.

Chapter 6 concludes the report. Some pointers to future extensions are provided.

CHAPTER 2

PIPELINED OBJECT TRACKING

In this chapter we have discussed the basics of pipelined object tracking and its algorithm. Detecting a particular object in each and every frame of the video is known as object tracking. Real time performance is the main challenge in object tracking. It depends on the algorithm used for tracking.

2.1 Object tracking

Detecting a particular object in every frame of the video is known as object tracking. Almost any existing object tracking algorithm can track an object, if the image sequences are noise free but the amount of resources it consumes may be huge. This section explains three different methods: (i) Background subtraction, (ii) Frame subtraction, (iii) Histogram of Oriented Gradients (HOG) for object detection in detail

2.1.1 Background subtraction

Background subtraction is the process of separating out foreground objects from the background in a sequence of video frames. Background subtraction is a widely used approach for detecting moving objects from static cameras. Fundamental logic for detecting moving objects from the difference between the current frame and a reference frame, called *background image*. Background being static means, there should not be any illumination changes, any moving leaves due to wind or any waves or ripples in pond, and any moving clouds. So the environment is much tighter as shown in the figure 2.1. Artificial lighting environment like inside an office may be an ideal static background.

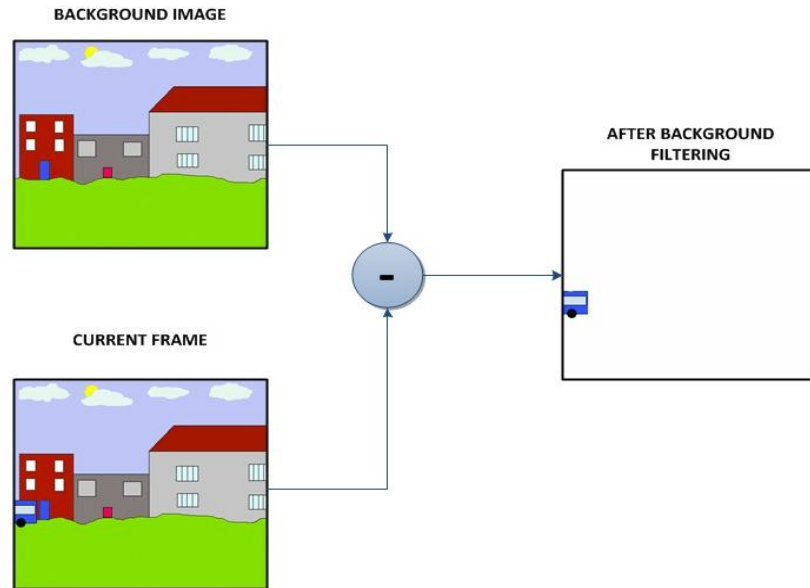


Figure 2.1: Background Subtraction

Gaussian mixture models, Eigen model etc. are used when background remains static. In Gaussian mixture models and Eigen models, the static background is modeled using a set of video frames. After 10 to 20 frames, objects can be tracked with ease using these background models. These background modeling algorithms handles changes in the background to some extent as it always adapts itself after every 10 to 20 frames. Every frame is subtracted from the background model to get a difference image. Using this difference image, all the foreground objects can be identified. From this foreground we can detect the required object using some other algorithms and hence the object can be tracked. Flow chart of Background subtraction is shown in the figure 2.2.

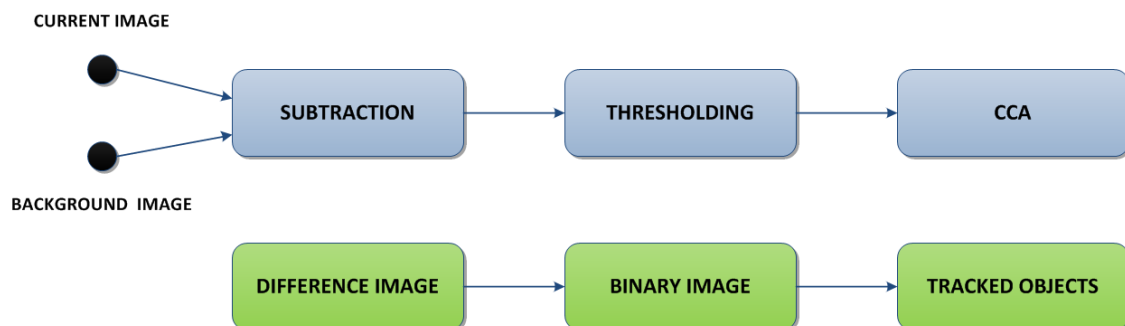


Figure 2.2: Flow chart of Background subtraction

In simple background subtraction, we will usually have the background of the video in advance. For example, inside an office, background can be an image (when nobody is present inside) with all lights switched on. There is another method called Frame subtraction, which is much sophisticated than Background subtraction as the variation in the background can be tolerated to some extent. Once we obtain the difference image, it has to go through *threshold* to obtain a binary image. Threshold is fixed for a particular video clip. It is easy to identify an ant in white background rather than a black background. It will also vary if you resize the images in the video. So choosing the right threshold is empirical.

2.1.2 Frame subtraction

Frame subtraction is an improvement made on the background subtraction where we subtract the n^{th} frame from $(n-k)^{\text{th}}$ frame instead of one frame, generally $k=1$. It works better than background subtraction as small movements in the background such as moving clouds, leaves and ripples in the pond etc. can be handled well. Background model has to be updated in each and every frame to handle such dynamics in the background. In the code that is developed based on the paper does not involve background modeling as in the paper and also does not handle occlusion. Flow chart of Frame subtraction is as shown in figure 2.3.

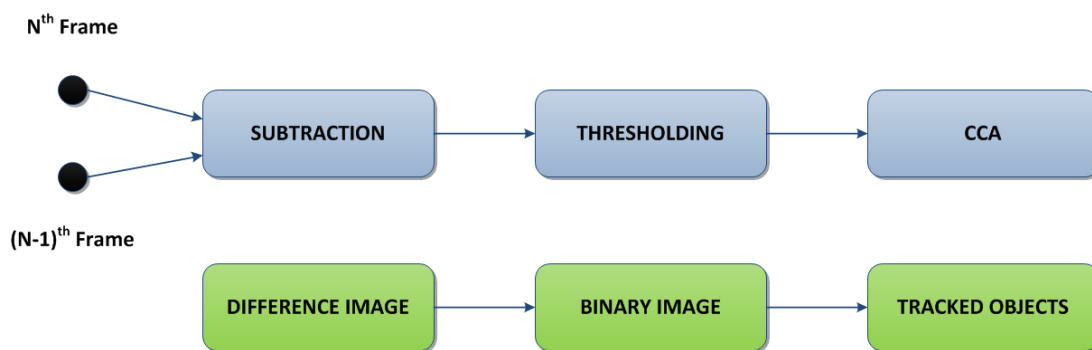


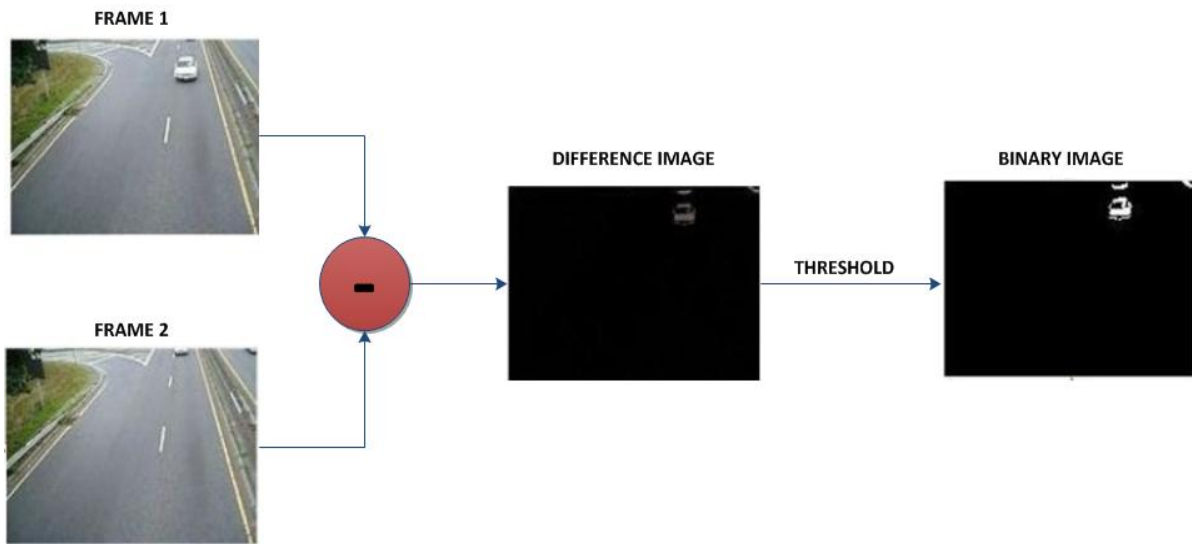
Figure 2.3: Flow chart of Frame subtraction

The difference image obtained in frame subtraction is quite different from that of background subtraction. Figure 2.4 shows how Frame subtraction and binarisation is done. The moving objects whose intensity is constant won't be visible in the binary image. If a square with constant

intensity (white) moves on a black background, we will be able to see only the edges of the square in the frame subtracted difference image. Whereas in background subtraction, you will be able to get complete square in the binary image. In reality such constant intensity objects are very rare and the noise present in the image will definitely turn on few pixels between the two lines.



(a)



(b)

Figure 2.4: Frame subtraction and binarisation

In frame subtraction method, we will essentially increase noise while subtracting two different frames. Hence the path of any object will be predicted with greater noise. If the object is moving in a complex track, the predicted track may not be smooth. Kalman filter can be used to reduce such noise and smooth track can be obtained in some cases.

Pseudo code 1: Frame subtraction and binarisation

```
1: Input : RGB 565 image
2: Output : Binary image
3:
4: for i = 1 to height(Image) do
5:     for j = 1 to width(Image) do
6:         grn_pix_img1= (0x07E0 & row_buffer_img1[j])>>5
7:         grn_pix1_img2 = (0x07E0 & row_buffer_img2[j])>>5
8:         if grn_pix_img1 > grn_pix_img2 then
9:             diff = grn_pix_img1-grn_pix_img2
10:        else
11:            diff = grn_pix_img2-grn_pix_img1
12:        end if
13:        if diff > binThreshold then
14:            bin_img_pix=0xFFFF
15:        else
16:            bin_img_pix=0x0000
17:        end if
18:    end for
19: end for
```

There are three more parameters which has relevance to getting nice blobs in the binary image. They are area threshold, pixel level threshold and subtracting frame (k frames older than the current frame). Value of k can usually be kept 1. That is, we are subtracting the current frame with previous frame. Pixel level threshold depends mostly on the contrast between the foreground and background objects. Pixel level threshold value has to be higher than the noise intensity and lesser than foreground intensity value. Its value can be kept as 25 (obtained after experimenting with 5-6 videos). Area threshold depends mainly on the size of the image. It is directly proportional to the size of the image. If the value of area threshold is 30 for 120x160 image sequences, it will be 60-70 for 240x320 image sequences. All these parameters have to be set correctly for a new video for proper tracking.

2.1.2 Histogram of Oriented Gradients (HOG)

Histogram of Oriented Gradient (HOG) is an object descriptor. A set of features (color, intensity gradient, area, scale etc.) which helps in describing an object is known as *descriptor*. HOG is a descriptor mainly used for detecting humans in an image. This set of features is invariant to scale (to some extent). It mainly depends on two factors, one is the contrast between foreground and background and the other is the geometry of the object.

HOG captures the local features of an object much better than any of the other descriptors. A detailed description of HOG can be read from Dalal and Triggs [7]. An online lecture Shah [14] is also available on HOG which gives very clear explanation of the same.

Extraction HOG features

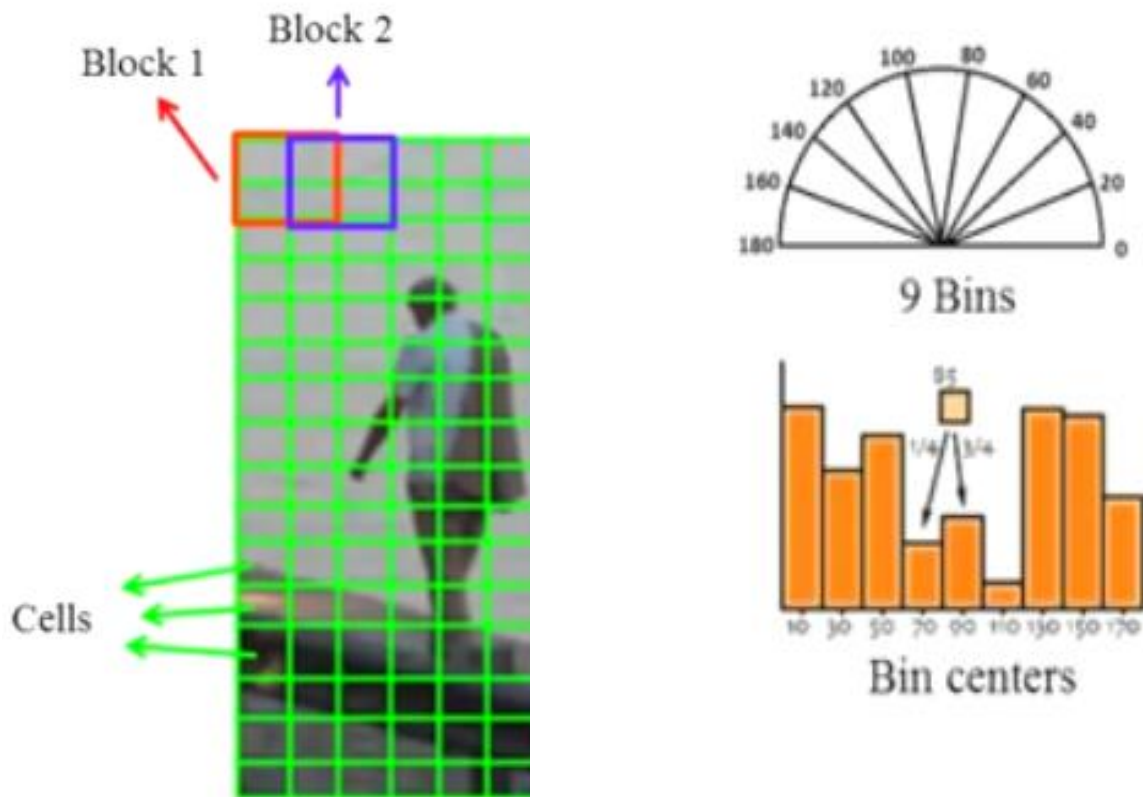


Figure 2.5: Extracting HOG feature

Figure 2.5: Histogram binning and interpolation

Consider the image shown in the figure 2.5. It is an image of size 64×128 . Divide the image into 16×16 blocks with 50 percent overlap. There will be a total of 105 blocks (7×15). Each block is further divided into 2×2 cells. A single cell has 8×8 pixels inside them. Take gradient of the cells and quantize them into 9 bins as shown in figure 2.6. This vector of length 9 is a feature vector describing that particular cell. Concatenating all the feature vectors gives us a super vector of size 3780 ($105 \times 4 \times 9$). This super vector describes the entire image. Authors of the paper Dalal and Triggs [7] have given a simplified code for extracting HOG feature from an image.

Tracking using HOG features

In the initial frame, we need to enter the object's location and its bounding box manually. From the initial bounding box, we extract HOG features and it describes the object for the next frame. We also extract HOG features from around the object in order to describe the background. In the next frame, we search for the object around its initial location. In the search radius we keep the size of the initial bounding box as constant. We keep extracting HOG features from the bounding boxes inside the search radius and compute its distance from the features extracted in the last frame. The bounding box which is closest to object feature (extracted in last frame) is assigned as the new location of the object in current frame.

During tracking, scale of the objects might vary through the image sequence and we need to predict the object's location in real time. Different objects will have different area on the image and hence we cannot use same window size for every object. Moreover, there is a problem of drift while tracking using descriptors. This drift occurs due to constant addition of noise from the background. Because of this drift, the bounding box slowly recedes from the object and gets stuck to one location in the background with slight jiggle from frame to frame. In the code, we have handled partial occlusion but it cannot detect an object which has disappeared due to complete occlusion and comes back again in the scene 20-30 pixels away from the original position. For example, it cannot detect a car completely occluded by a tree once and comes back into the scene at the other end of the tree. In order to detect the car in such situations, search radius has to be increased and it might affect real time performance.

In order to improve the feature vector of the object, we included initial location, mean, variance and skewness of intensity along with HOG features. Initial location was useful when the object is moving slow otherwise drift gets increased.

2.2 Issues in object tracking

Real time speed requirements, illumination changes, object shape changes, noise in images and occlusion are few major problems when dealing with object tracking. While we cannot remove noise which depends on the sensor used inside the camera we can deal with other issues to some extent. Handling occlusion is one of the major issues in descriptor based tracking. There are two different occlusions, partial and complete occlusion. To deal with occlusion, machine learning (or statistical) concepts can be used rather than simple algorithms. For example, if we want to track a particular book and we are able to track it using descriptor based object tracking. Certain unique features or descriptors are extracted from the book and used for detecting it in every frame. Now if some other book partially starts occluding, then the descriptor gets changed. We have two set of descriptors representing the same book. When the book comes out of occlusion, the algorithm might start tracking the other book which occluded our original book. These kinds of issues usually come up while doing experiments with different set of videos.

2.3 Connected Component Analysis (CCA)

Connected components analysis is a well-known pre-processing step in many image processing applications. It is not only used to divide the image into its constituent parts and to give different labels for each segment, but also a key step in the tracking of moving objects in video sequences. Image labeling using the connected components analysis is a key step in pattern recognition, target tracking, computer vision, fingerprint identifications, character recognition, medical images analysis and many other image-based applications. Connected component analysis can be defined as the process in which a binary image is transferred into N-state image where all connected pixels which belong to one object are assigned a unique label.

Connected component analysis is the main sub-module in frame subtraction algorithm. After thresholding the difference image, we obtain a binary image. This binary image has blobs which have to be analyzed to obtain foreground object's location in image, area and bounding box. In figure 2.7 there are three complex objects and they have to be labeled as shown. CCA helps us to obtain area, bounding box and centroid information from the binary image. CCA retrieves the features (area, bounding box etc.) in a single pass without giving any labels.

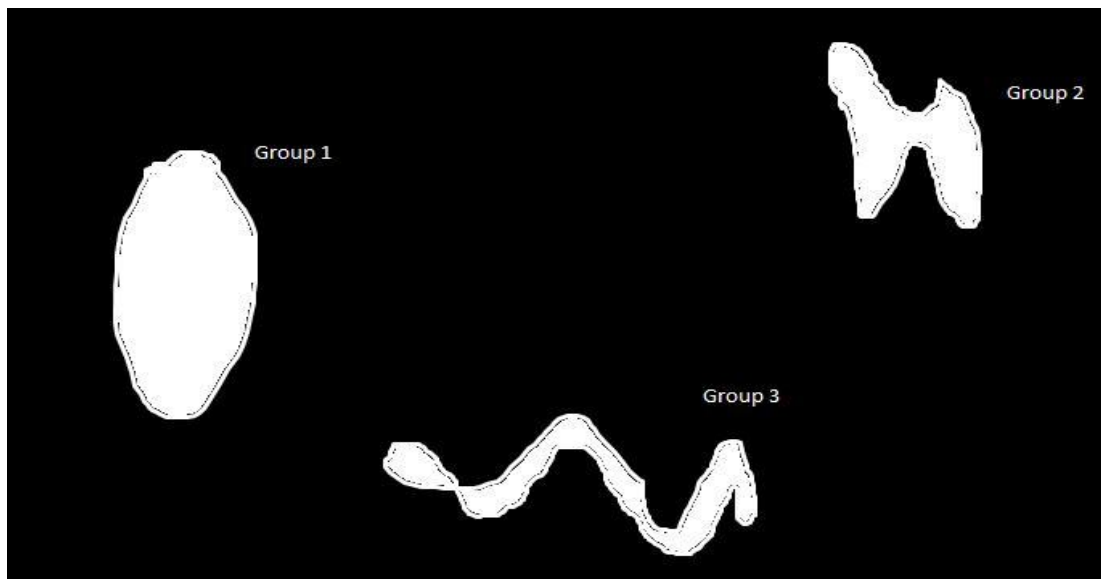


Figure 2.7: Connected components

Connected component labeling has been widely investigated and many algorithms have been proposed. These algorithms vary in their complexity and speed, and they can be roughly classified into two main categories. The first group of algorithms attempts to solve the connectivity between pixels using multiple scans. They keep scanning the image, forward and backward to resolve label equivalences until there is no change. The other group attempts to assign the labels using two-scans only. In the first scan they assign initial labels and in the second scan they resolve the equivalence between the labels. In this report, a fast two-scan connected component algorithm is implemented to divide the image into its individual components.

2.4 Single pass algorithm for CCA

The single-pass connected components algorithm has four main blocks, as shown in figure 2.8:

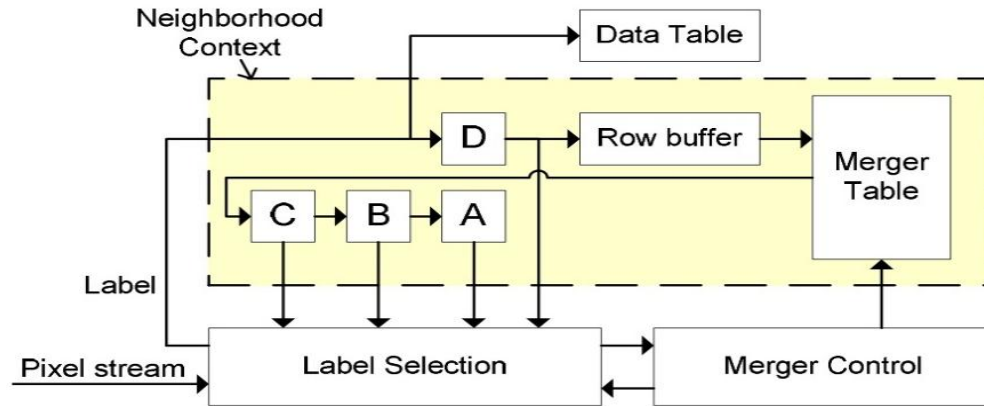


Figure 2.8: Basic architecture of Single pass algorithm.

1. The neighborhood context block provides the labels of the four pixels connected to the current pixel being processed.
2. The label selection block selects the label for the current pixel based on the labels of its neighbors.
3. A merger control block is required to update the merger table whenever two objects are merged.
4. The data table accumulates the raw data from the image required for calculating the features of each connected component. Since the image data is not retained, it must be accumulated for each connected component as the image is scanned.

2.4.1 Neighborhood context

The neighborhood context is implemented in much the same manner as a window filter. The neighboring pixel labels are stored in registers A, B, C, and D as show in figure 2.9. These are shifted along each clock cycle as the window is scanned across the image. Since the resultant labels are not saved, the labels from the

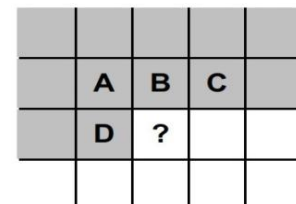


Figure 2.9: A label is assigned to the current pixel based on already processed neighbors.

previous row must be cached using a row buffer. The labels from the cache must also be looked up in the merger table to correct the label for any mergers since the pixel was cached.

2.4.2 Label selection

The label for the current pixel follows that of the classic connected components algorithm:

- Background pixels are given a label of 0.
- If all of the neighboring pixels are background, then a new label is assigned.
- If only a single label is used among the labeled neighbors, that label is also assigned to the current pixel.
- If two different labels are used among the neighbors, then this indicates a merger condition. For reasons that will be explained later, we assign the smaller of the two labels to the current pixel. In practice, the selection logic may be simplified by considering the neighborhood pixels in a particular order. A merger will only occur if pixel B is background and the label of C is different from either A or D. In all other cases, the labels will already be equivalent from prior processing.

2.4.3 Merger control

The merger table is used as a look-up table on the output of the row-buffer. This ensures that the correct label is used for any labels that have been stored in the row buffer which have subsequently been merged. Whenever a new label is created, a new entry is added to the merger table pointing to itself. This avoids the need for initializing the merger table prior to processing. To keep the merger table up to date, whenever a merger occurs, the label that is replaced should subsequently point to the merged label. This means that in addition to reading from the merger table every clock cycle, there is also the need to write on some clock cycles. Since only one access may be made to memory each clock cycle, this requires either running the merger table RAM at twice the clock frequency, or using dual-port RAM with one port for the read and one for the write. When a merger occurs, the new label must replace all occurrences of the old label within the neighborhood. This requires a multiplexer on the input of register B. Register A does

not require a multiplexer because B is always a background pixel whenever regions are merged. Timing considerations also require a multiplexer on the input of register C. The next value for C is being read at the same time that the merger table is being updated. Therefore the value read from the merger table will reflect the label before the merger. So if the next value for C is not the background label, the newly merged label should also be copied to C. It is essential that the correct label be selected to represent the region when a merger occurs

2.4.4 Data table

As each pixel is processed, the data table is updated to reflect the current state of each region. The data that needs to be maintained depends on the features that are being extracted from each connected component. If only the number of regions is required, this can be determined with a single global counter. Each time a new label is assigned, the counter is incremented, and when two regions merge, the counter is decremented. For measuring the area of each region, a separate counter is maintained for each label. When two regions are merged, the counts are combined. For determining the center of gravity of the region, the sums of the x and y coordinates are maintained. At the end of the image, these can be divided by the area to give the region center. To obtain the bounding box of each region, the extreme coordinates of the pixels added are recorded. Other, more complex, features may be determined in a similar manner by accumulating the appropriate raw data.

2.4.5 Table sizes

Both the merger and data tables require one entry for each label used. The worst case for the number of objects possible in an $M \times N$ image is illustrated in Figure: 2.10 and requires

$Number = \text{ceil}\left(\frac{M}{2}\right) \times \text{ceil}\left(\frac{N}{2}\right)$ Objects. In practice the actual number of labels required is significantly less than this, with the number required depending on the expected region size and shape.

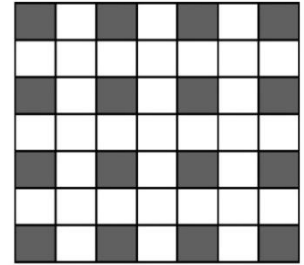


Figure 2.10: Worst case number of objects

Pseudo code 2: Single pass connected component analysis

```
1: Input : Binary image
2: Output : Data tables storing component features
3: Initialisation : a=0, c=0, c=0, d=0, label_cnt=0
4: Row_buffer[i]=0 for all i
5: Merger_table[i]=0 for all i
6: Left_border_table[i]=0 for all i
7: Right_border_table[i]=0 for all i
8:
9: for i = 1 to height(Image) do
10:   for j = 1 to width(Image) do
11:     p = pixel(i, j)
12:     if p is not object_pixel then
13:       current_label = 0
14:     end if
15:     if p is object_pixel then
16:       if each of a, b, c, d = 0 then
17:         label_cnt = label_cnt + 1
18:         current_label = label_cnt
19:       end if
20:       if exactly one of a, b, c, d  $\neq$  0 then
21:         current_label=non_zero_value(a, b, c, d)
22:       end if
23:       if c and a  $\neq$  0 or c and d  $\neq$  0 then
24:         current_label = minimum(c, a) or minimum(c, d)
25:         Merger_table[c] = current_label
26:         Merger_table[a] = current_label
27:         Merger_table[d] = current_label
28:         update Data_table[current_label]
29:       end if
```

```

30:          if j = 1 then
31:              Left_border_table[i]=current_label
32:          end if
33:          if j = width then
34:              Right_border_table[i]=current_label
35:          end if
36:      end if
37:      a = b
38:      b = c
39:      Row_buffer[j-1] = d
40:      d = curren_label
41:      c = Row_buffer[j+2]
42:  end for
43: end for

```

SUMMARY

1. Real time object tracking can be accomplished via parallel processing or multithreading and pipelining.
2. The main aim of any object tracking algorithm is to provide the trajectory of an object over time by locating its position every frame of the video.
3. In simple background subtraction, we will usually have the background of the video in advance.
4. Handling occlusion is one of the major issues in descriptor based tracking.
5. Frame subtraction is an improvement made on the background subtraction where we subtract the nth frame from $(n-k)^{\text{th}}$ frame instead of one frame.
6. Connected components analysis is used to divide the image into its constituent parts and to give different labels for each segment, but it is also a key step in the tracking of moving objects in video sequences.

CHAPTER 3

IMPLEMENTATION ON MULTICORE XMOS ARCHITECTURE

This chapter gives an introduction to experimental setup. It explains the parallel processor we used in this project and also gives an idea of how the setup has been developed. Experimental results are provided here. Real time operation can be accomplished via parallel processing or multithreading and pipelining. A thread is a part of a program which can run independently. Any process or an application program can be split into independent smaller codes which can be made to run as a thread. Multithreading CPUs have hardware support to run each thread concurrently. It can be compared with multiprocessing. In object tracking process, we have memory interface, algorithmic unit and LCD display for the output as independent units of the code.

3.1 SliceKIT and setup

SliceKIT comprises a core board powered by an xCORE multicore microcontroller with four slots for plugging in four I/O sliceCARDS. The sliceKIT core board features a dual-tile device that delivers the deterministic, responsive processing required to handle a variety of peripheral interfaces, data processing and control tasks. Each additional I/O sliceCARD is supplied with a demo application that allows getting up and running quickly. The result is a framework of peripherals and I/O providing with an exact fit chip for your system.

A large and growing range of expansion slices supporting a broad range of I/O types are available. The slices connect to the core board using low cost PCIe style connectors. This reduces the cost of slices since the connectors are simply contacts on a dual sided PCB. It also means adding slice is straightforward. Core board comprises of 16 core xCORE multicore microcontroller. Rapid prototyping of systems becomes possible with slice kit because it supports a lot of interfaces available as slices. We use SDRAM and LCD slices for our project. The experiment setup is shown in the figure 3.1.



Figure 3.1: Slice kit setup

SDRAM used in our project has a size of 8 Megabytes. LCD screen size is 480 x 272. In the main board you will be able to see four slots available: square, circle, star and triangle. We can even interchange the position of LCD and SDRAM but we need to modify ports accordingly. As discussed, before developing applications for slice-kit we need to understand its hardware specifications.

In the figure 3.2 socket_00, socket_01, socket_10, socket_11 represent square, star, circle and triangle respectively. X0 indicates XCore0 and X1 indicates XCore1. We have to use correct ports in the application program.

CORE BOARD											
Connector	PLUG_00		SOCKET_00		SOCKET_01		SOCKET_10		SOCKET_11		
Core	0		0		0		1		1		
Type	Reverse 0		0		1		0		1		
Side	B (Top)	A (Bottom)	B (Top)	A (Bottom)	B (Top)	A (Bottom)	B (Top)	A (Bottom)	B (Top)	A (Bottom)	
Pin Number	1	DEBUG	MSEL	NC	NC	NC	NC	DEBUG	MSEL	NC	NC
	2	TCK	NC	NC	5V	X0D0	5V	TCK	5V	X1D0	5V
	3	GND	TMS	GND	NC	GND	X0D12	GND	TMS	GND	X1D12
	4	TDO	TDI	NC	NC	X0D11	X0D23	TDI	TDO	X1D11	X1D23
	5	PRSENT	GND	3V3	GND	3V3	GND	3V3	PRSENT_N	3V3	GND
	6	X0D9	X0D3	X0D2	X0D8	X0D26	X0D32	X1D2	X1D8	X1D26	X1D32
	7	X0D8	X0D2	X0D3	X0D9	X0D27	X0D33	X1D3	X1D9	X1D27	X1D33
	8	GND	X0D10	GND	X0D1	GND	X0D25	GND	X1D1	GND	X1D25
	9	X0D7	X0D5	X0D4	X0D6	X0D28	X0D30	X1D4	X1D6	X1D28	X1D30
	10	X0D1	GND	X0D10	GND	X0D34	GND	X1D10	GND	X1D34	GND
	11	X0D6	X0D4	X0D5	X0D7	X0D29	X0D31	X1D5	X1D7	X1D29	X1D31
	SLOT										
	12	X0D21	X0D15	X0D14	X0D20	X0D36	X0D42	X1D14	X1D20	X1D36	NC
	13	X0D20	X0D14	X0D15	X0D21	X0D37	X0D43	X1D15	X1D21	X1D37	NC
	14	CLK	GND	CLK	GND	CLK	GND	CLK	GND	CLK	GND
	15	X0D13	X0D22	X0D22	X0D13	X0D24	X0D35	X1D22	X1D13	X1D24	X1D35
	16	GND	RST_N	GND	RST_N	GND	RST_N	GND	RST_N	GND	RST_N
	17	X0D19	X0D17	X0D16	X0D18	X0D38	X0D40	X1D16	X1D18	X1D38	NC
	18	X0D18	X0D16	X0D17	X0D19	X0D39	X0D41	X1D17	X1D19	X1D39	NC

Figure 3.2: Slice kit port map

3.2 LCD Slice

Features

- 480 x 272 full color display
- 40-pin ZIF connector with ribbon cable to the display
- Resistive touch screen with 2 wire interface to xCORE

The LCD used in our project has a resolution of 480 x 272 with pixel format of RGB565. The xCORE multicore microcontroller drives the display directly without the use of an external LCD controller, via a parallel RGB interface. The SDRAM slice could also be a useful addition to system, providing additional data memory for frame buffering. We have used `sc_lcd-master` which is an open source code of XMOS available at <http://www.github.com/xcore>. It displays XMOS logo on the screen. In this case, they store the entire image in RAM to display the image and use RGB565 format for lightening each pixel. The main reason for not using RGB888 format is due to lack of availability of output pins in the slice kit.

The code writes two neighboring pixels at a time. An integer data type is of 32 bits, first 16 bits correspond to one pixel and the later 16 bits correspond to its neighboring pixel. For example, 0xFFFF0000 makes the first pixel black and second pixel white. An image has to be in above format to be displayed on the LCD screen. First an image of RGB888 format has to be converted into RGB565 format. Then we need to concatenate two neighboring pixel. The following example explains how to concatenate two pixels. The figure 3.3 gives overall picture of the SDRAM working.

0xF0FF 0x00FF 0x01AE 0xFF0F . . . be the first four pixel value in RGB565 format without concatenation. 0x00FFF0FF 0xFF0F01AE will be our new concatenated format useful for displaying on LCD screen. So an image of size 120x160 in RGB24 format will become 120x80 in our new format.

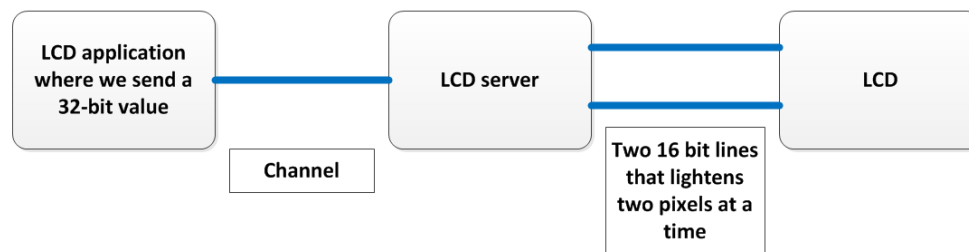


Figure 3.3: LCD server

LCD server takes care of the port configuration. It gets the data to be displayed in the correct format from the application program written by us and displays it on the screen. `lcd_init`, `lcd_req`, `lcd_update` are three main functions used in the code. `lcd_init` checks whether the LCD is ready for use. It checks for a control token of a given value. If the next byte in the channel is a control token which matches the expected value then it is input and discarded, otherwise an exception is raised. `lcd_req` is again similar to `lcd_init` where a token is sent through channel indicating the previous write operation on the LCD is complete and hence the server can receive next data to be written on the screen. `lcd_update` receives a row of information from the application program (row here means 240 x 32 bits or an integer array of size 240. Number of pixels in a row is 480. Since we write two pixels at a time in RGB565 format we only need an integer array of size

240). LCD server takes care of how to display the entire row. Having finished displaying the entire row, `lcd_req` will receive a token indicating that we can send the next row's data. This happens in an infinite while loop to display an image continuously.

Double buffer

Double buffer is used to prevent flickering while displaying image on the LCD. We have image data stored in SDRAM. As the size of the LCD screen is 480 x 272, a single row of pixel will need a buffer of size 240 x 4 bytes. An integer array of size 240 is required to display one row in LCD. It is 240 and not 480 because we use RGB565 format.

In double buffer usage, we have two integer buffers of size 240. While buffer A is being displayed on the LCD, buffer B is getting filled up with data for next row of pixels from SDRAM. Timing becomes important for real-time performance. In the code given in appendix, while displaying buffer A, we are clearing buffer B instead of loading it with next row data and vice versa. The necessity of clearing will be explained shortly. We can also triple buffer if we have memory to improve on time complexity.

There are two other functions that are fairly important. "add" and "sub" function. The server works with a dual buffer concept. There are two integer buffers of size 240. Buffer A is meant for displaying even rows while buffer B is meant for displaying odd rows on the LCD screen. The total number of rows on the LCD is 272. Let the size of an image we need to display be 120 x 160 and assume we display it in the left corner of the LCD. Both A and B are initialized with the background color to start with. Between 1st row and 120th row we need to update the values of these buffers according to the image and this is done by the "add" function. Once we have finished displaying the entire image, we must display background for the entire 121st row but in the buffer B we will still have values of 119th row's data. Hence, the rest of the rows (121 - 270) will be filled with values of 119th row. To avoid this we use sub function which updates (resets) the buffers with background color after displaying the entire image.

SDRAM and LCD are running in parallel. The following code snippet will help you understand they run in two different threads. Even though only 2 threads are required, 6 more threads are added to simulate worst case scenario. Instead of `par (int i=0; i<6; i++) while(1);`, we can write

while(1); 6 times both means the same. The figure 3.4 shows the clock frequency versus number of threads. If more than 4 threads are active, each thread is allocated at least $1/n$ cycles (for n threads).

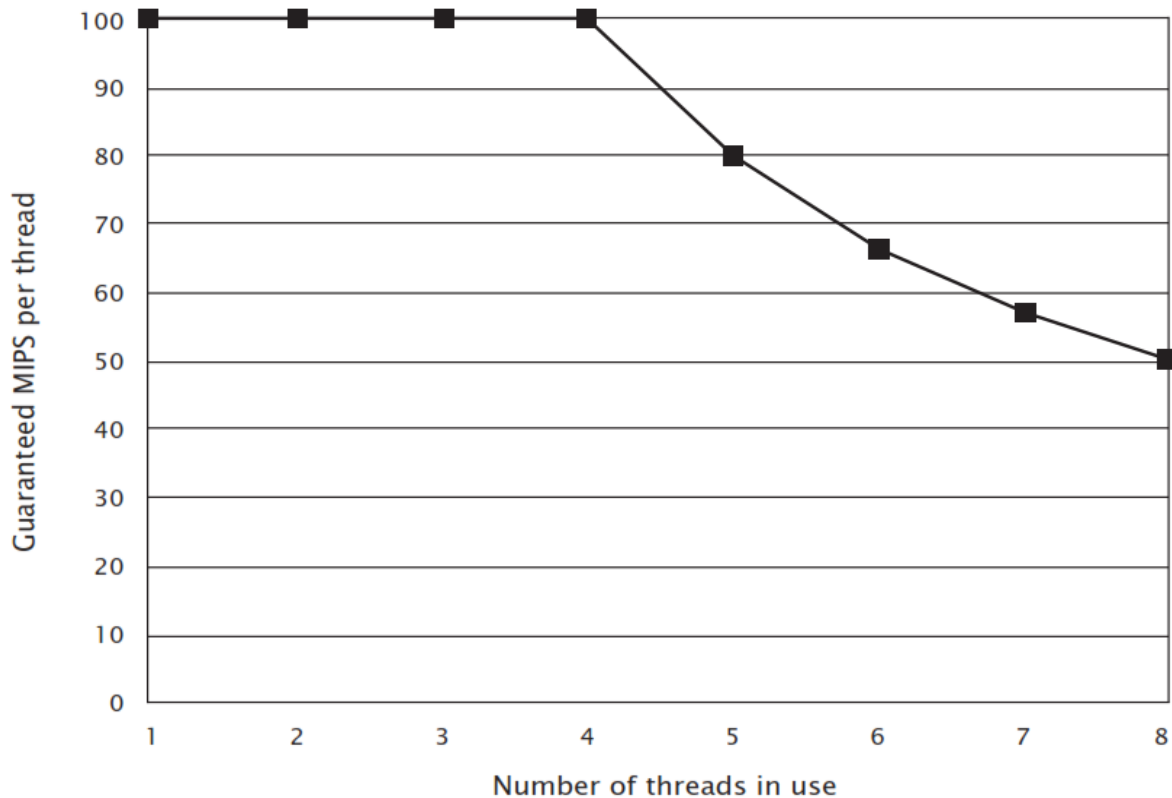


Figure 3.4: Processing speed versus number of parallel threads

3.3 SDRAM Slice

The SDRAM Slice card provides a large external random access memory to sliceKIT, perfect for any application where a large external data buffer is required: image buffering for a display controller, audio buffering for effects processing, or perhaps data capture, processing and logging.

Features

- 8 MByte SDRAM
- Clock speed upto 50MHz
- Data rate upto 80MBytes/second

The memory available in SDRAM is split into 4 banks. Each bank has 212 rows, 28 columns of cells where a cell is a 16-bit memory. We have used `sc_sdr_burst-master` an open source code developed by XMOS available at Jason. The figure 4.4 gives overall picture of the SDRAM working. There are three main functions used for performing writing and reading operations in the SDRAM. They are

1. `sdr_buffer_write(chanend, bank, row, col, size, buffer)`
2. `sdr_buffer_read(chanend, bank, row, col, size, buffer)`
3. `sdr_wait_until_idle(chanend, buffer)`

‘chanend’ is one end of the channel used for communication between the application program and the sdr server. Bank can take any value between 0 and 3 choosing one among the four banks available. Similarly row and col are used to address a specific location in the SDRAM. ‘size’ is the size of the buffer that we want to be written onto or read from SDRAM.

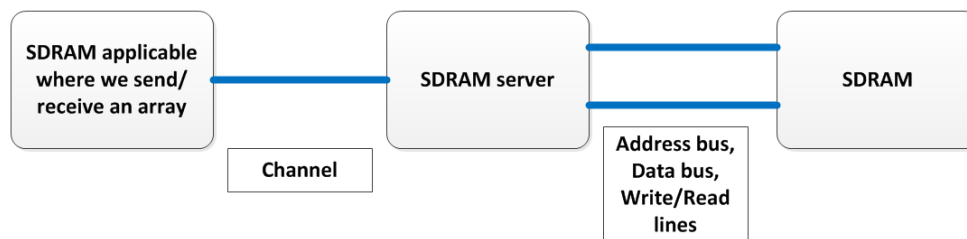


Figure 3.5: SDRAM server

The functions read/write two cells at a time. This can only be understood with an example. Say we need to write an integer array of 3 words.

Buffer = {0xFFFF, 0x1FFF, 0x2FFF}

Invoking the 1st function with size equals 3, `sram_buffer_write(chanend, 0, 0, 0, 3, Buffer);` sram cells are written in the following fashion.

0x0000	0xFFFF	0x0000	0x1FFF	0x0000	0x2FFF
--------	--------	--------	--------	--------	--------

Table 3.1: Buffer array as stored in sram

Now invoking the 2nd function with “col” changed as “col+1”, `sram_buffer_read(chanend, 0, 0, 1, 3, Buffer);` will produce a buffer with following values.

Buffer = {0xFFFF0000, 0x1FFF000, 0x2FFF****}

3.4 xTIMEcomposer Studio

xTIMEcomposer Studio is based on the industry standard Eclipse IDE. Additional perspectives and views have been added to support the unique tools that XMOS has developed to make it easy to develop real-time embedded applications for xCORE multicore microcontrollers. They include:

- XMOS Timing Analyzer, which uses the determinism of the xCORE architecture to deliver verified time-critical solutions;
- xSCOPE for real-time non-intrusive code instrumentation and debugging;
- xSOFTip Explorer for browsing our library of soft peripherals and code blocks.

3.5 Thread Scheduling

Object tracking app is divided in to three phases

Phase 1: Loads TGA images from hard disk to SDRAM with the help of loader. RGB565 values of pixels in each row are written to SDRAM.

Phase 2: Frame subtraction and binarisation using given threshold. Single Pass CCA reads the image again row by row, finds bounding boxes of connected components. The image is read once again for annotation by app. The annotated image is written back to SDRAM. The threshold computation, CCA and annotation can be pipelined for successive images.

Phase 3: The annotated images are displayed one after another to display video.












Thread	Phase 1	Phase 2	Phase 3
Loader			
Demo App			
Frame subtraction + Binarisation			
Single Pass CCA			
Annotate			
Display Manager			
Display Controller			
SDRAM server			
LCD server			

Table 3.2: Thread Scheduling

3.6 Implementation

While implementing such an algorithm in XMOS embedded platform, the image data (RGB565 format) is streamed from the SDRAM in a raster format. Classical labeling algorithm is a two pass labeling process which requires the whole image to be present in the buffer to extract features. Unfortunately, memory available in XMOS is as low as 64kb/core. So an optimized single pass connected component analysis is implemented instead of classical connected component analysis.

3.6.1 Image format

Image format used is RGB565 format. We read image.tga file of size 480 X 272. In RGB888 format, we need 382.5kilobytes ($480 \times 272 \times 3 \div 1024$) to store image.tga. The image size is much more than 64kilobytes. We need to break the image into 8 parts and load them part by part

into SDRAM. Each part is 47.8125kilobytes. The file image.tga has 8 bits each for red, blue and green. We need to convert it to RGB565 format by ignoring 3 least significant bits from blue and red, and 2 least significant bits from green.

The RGB565 format image obtained is loaded into SDRAM. We use only green component of the image in connected component analysis. Green is less noisy when compared to red and blue since we lose only 2 least significant bits. A RGB image when converted to gray scale, green has about 70 percent contribution. RGB565 format is used to brighten each pixel in the LCD. This is shown in figure 3.6.

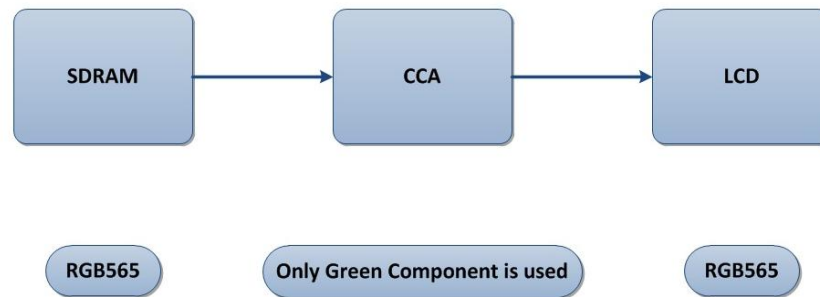


Figure 3.6: Image format

3.6.2 Pipelined Object tracking system

The pipelined object tracking implemented is shown in figure 3.7 below. There are three pipelined stages: (i) Frame subtraction and binarisation, (ii) Connected component analysis, (iii) Annotation. In annotation we draw green bounding box for an object. Since there is so much noise in binary image due to illumination changes and object shape changes we have to fix object threshold. Object threshold depends on moving object and illumination changes.

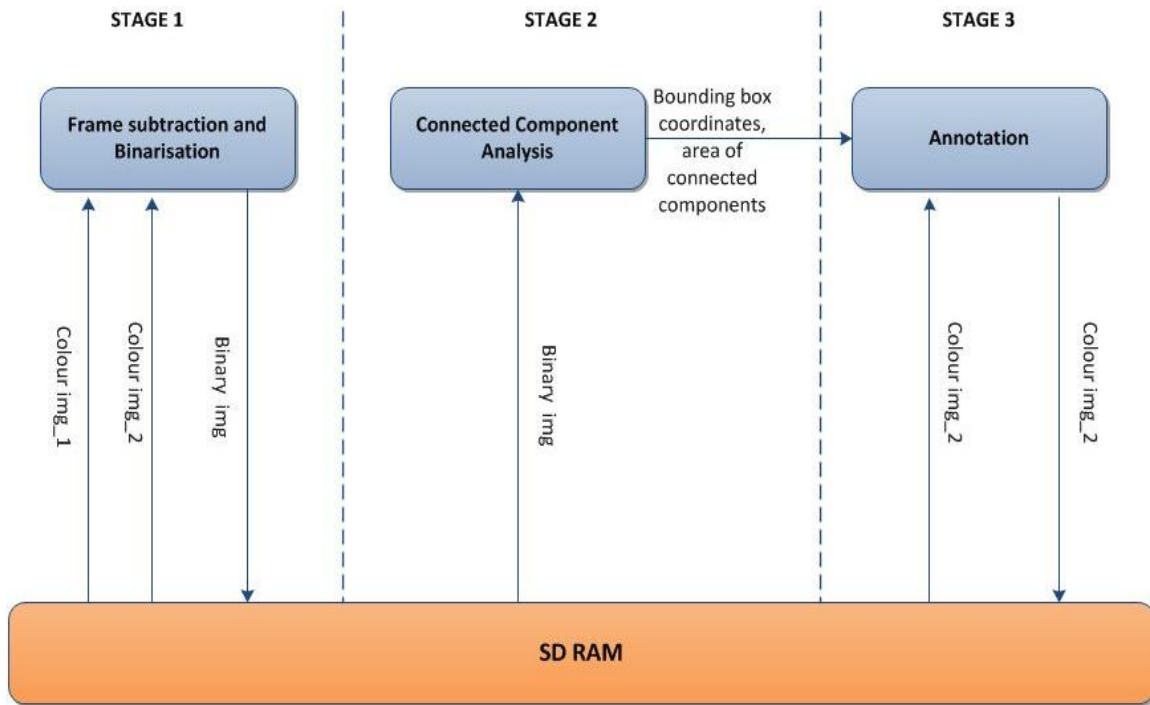


Figure 3.7: Pipelined object tracking

The design runs 6 threads in parallel. The figure 3.8 will describe how the threads interact with each other and display tracked object in the LCD.

In figure 3.8 `c_loader`, `client`, `c_sdram` and `c_lcd` are channels used for communication between different threads. The grey rectangular boxes `app`, `loader`, `display_controller`, `sdram_server` and `lcd_server` are threads. They all run in parallel. The green rectangular boxes are hardware that is connected to the slice kit. SDRAM corresponds to SDRAMslice and LCD corresponds to LCD slice. `Sdram_ports` and `lcd_ports` are specific ports to which sdram slice and lcd slice are connected to slice kit.

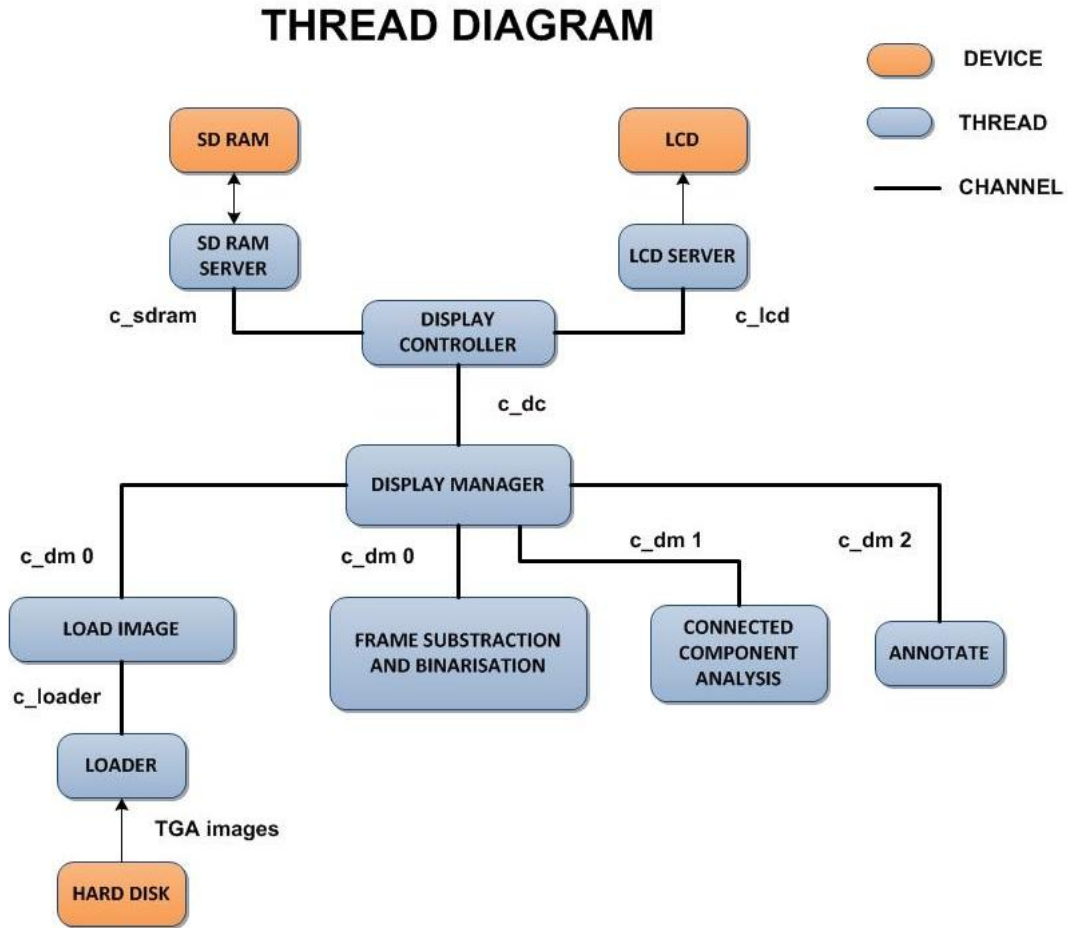


Figure 3.8: Thread diagram

Loader thread does the following steps until all the images are loaded into the SDRAM.

1. Read image in TGA format of size 480 x 272
2. Divide them into 8 parts
3. Load each part into SDRAM after converting each pixel to RGB565 format (It concatenates two pixels into one 32-bit integer)

The app thread in figure 3.7 is our object tracking application. It takes image data from SDRAM, does the processing and makes necessary changes to the image data to display it in LCD along with the bounding box around the object. In detail, app thread reads one row of pixel data (480 pixel = integer array of size 240) from current image and same numbered row from previous frame. It subtracts these two arrays to get a difference array. Difference array needs to be processed pixel by pixel to obtain a binary image. We take 1 variable from integer difference

array of size 240. An integer variable occupies 32 bits while a pixel occupies only 16 bits in RGB565 format. There are two pixels concatenated into one single integer variable. As described earlier, 1st 16 bits of the integer corresponds to pixel 1 and later 16 bits corresponds to pixel 2 of the LCD.

We need to separate red, green and blue from 16 bit data. As per the format, 5 least significant bit corresponds to red, 5 most significant bit corresponds to blue and the remaining 6 bits in the middle corresponds to green. We consider only green component of the difference array. The green component is multiplied by 4 and passed on to thresholding. After thresholding the pixel (green difference of the pixel), it is passed on to optimized CCA. Once all the pixels (480 x 272) are processed, we get area, bounding box and centroid information of the object from optimized CCA. Using the extracted features, we can track the object. Display controller thread `c_dm` manages SDRAM server and LCD server. It manages reading and writing of data into sdram. Next image handle is maintained in such a way that overwriting does not occur on lcd. Further information is given in comments section of the code. Display controller is available online Jason [10].

3.7 Results

The results obtained from object tracking application are shown in figures below



Figure 3.9.1: Bounding box has been put over the moving object in frame 1



Figure 3.9.2: Bounding box has been put over the moving object in frame 2

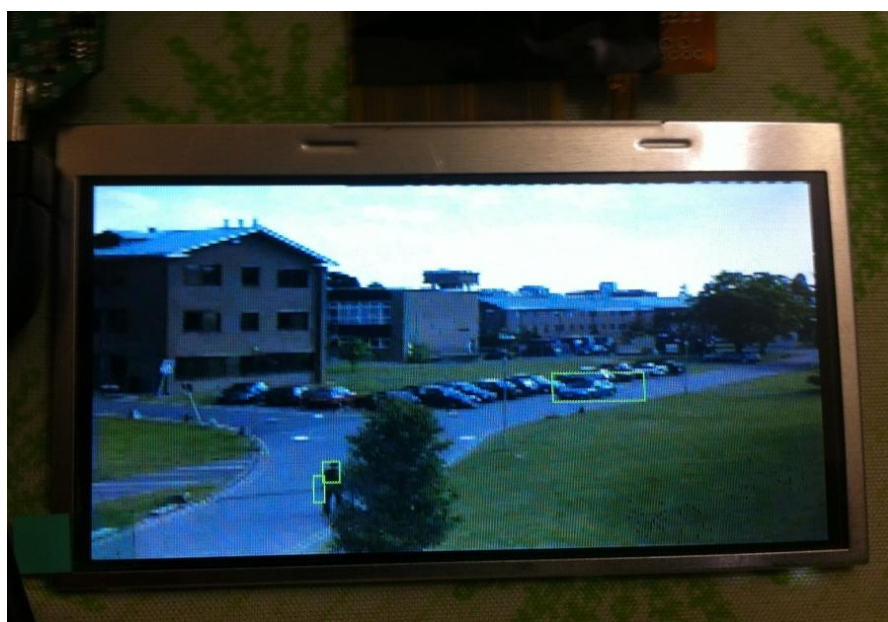


Figure 3.9.3: Bounding box has been put over the moving object in frame 3



Figure 3.9.4: Bounding box has been put over the moving object in frame 4



Figure 3.9.5: Bounding box has been put over the moving object in frame 5



Figure 3.9.6: Bounding box has been put over the moving object in frame 6



Figure 3.9.7: Bounding box has been put over the moving object in frame 7



Figure 3.9.8: Bounding box has been put over the moving object in frame 8

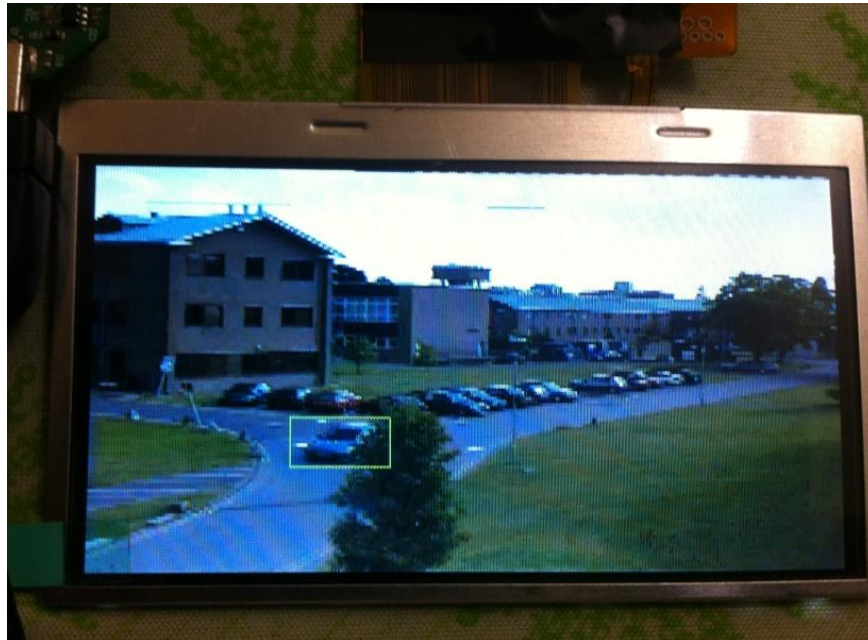


Figure 3.9.9: Bounding box has been put over the moving object in frame 9

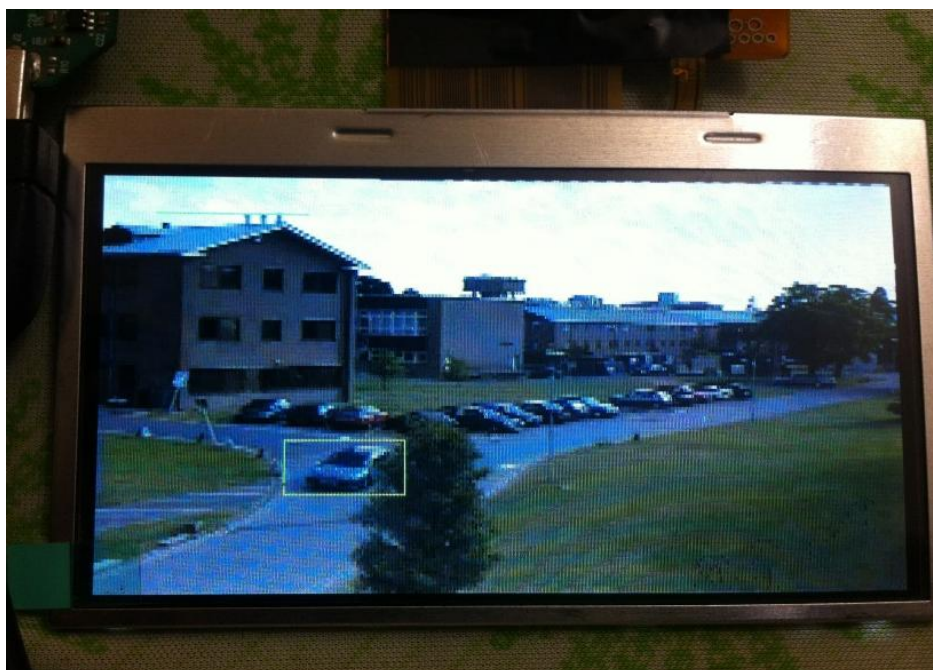


Figure 3.9.10: Bounding box has been put over the moving object in frame 10



Figure 3.9.11: Bounding box has been put over the moving object in frame 11

SUMMARY

1. SliceKIT comprises a core board powered by an xCORE multicore microcontroller with four slots for plugging in four I/O sliceCARDS.
2. LCD has a resolution of 480 x 272 with pixel format of RGB565.
3. Double buffer is used to prevent ‘flickering’ while displaying image on the LCD.
4. The 8 MByte memory available in SDRAM is split into 4 banks. Each bank has 212 rows, 28 columns of cells where a cell is a 16-bit memory.
5. xTIMEcomposer Studio is based on the industry standard Eclipse IDE used to develop real-time embedded applications for xCORE multicore microcontrollers.

CHAPTER 4

MEMORY AND TIMING ANALYSIS

In this chapter we have discussed Resource utilization, memory analysis and timing analysis in detail. Memory in embedded systems can be categorized as program memory, stack memory and free memory.

4.1 Memory analysis

Memory analysis is important in embedded systems in order to design an application with lowest memory consumption. In C language, we have memory consumption for various data types as per Table 4.1

DATA TYPE	MEMORY
Integer	4 bytes
Char	1 byte
Signed char	1 byte
Float	4 bytes
Double	8 bytes

Table 4.1: Memory required for different datatypes

When we build a XC code, the compiler converts it to a binary file. This binary file is loaded into XMOS processor to run the application in XMOS hardware. XMOS processor has an internal RAM memory of 64 kilo bytes. The compiler checks whether the memory usage doesn't exceed 64 kilo bytes while building the XC code. Sum of program memory, stack memory and free memory will always be equal to 64 kilo bytes. Stack memory consists of variables such as integers, characters, arrays etc. Program memory consists of syntax and logic.

In X MOS development environment, double click on the binary file available in the project explorer. This will open a window consisting of memory consumption, number of logical cores used and number of timers used. All the experimental data given in this chapter is obtained from X MOS development environment. This helps programmer to know which part of the code goes to which type of memory.

4.2 Space Complexity (Processor memory usage)

Different phases of object tracking App are explained in last chapter

Operation	Phase 1	Phase 2	Phase 3
Loader	$O(w+n)$		
Load image	$O(w)$		
Frame subtraction + Binarisation		$O(w)$	
Single Pass CCA		$O(w+c)$	
Annotation		$O(w)$	

w – Width of the image, n – number of images, c – number of connected components

Table 4.2: Space complexity

4.3 Image Processing Operations - I/O and Memory

Loader

Inputs: TGA file names

Outputs: RGB565 values of image pixels (via c_loader)

Memory: To store file names, RGB565 values of pixels of one row of image

Load image

Inputs: RGB565 values of pixels (via c_loader)

Outputs: RGB565 values of each row of image to SDRAM

Memory: To store RGB565 values of pixels of one row of image

Frame Subtraction and binarisation

Inputs: RGB565 values of each row of image from SDRAM

Outputs: Binary image

Memory: To store RGB565 and binary values of one row of image

Single pass CCA

Inputs: RGB565 values of each row of image from SDRAM

Outputs: Number of connected components, bounding box coordinates, area, center of gravity.

Memory: To store merger table and bounding box coordinates in arrays of size equal to the maximum number of connected components, previous row buffer

Annotation

Inputs: Number of connected components, bounding box coordinates, RGB565 values of each row of image from SDRAM

Outputs: RGB565 values of annotated row of image to SDRAM

Memory: To store RGB565 values of pixels of one row of image

4.4 Resource utilization

The figure 4.10 shows the resources utilized for object tracking application. In sliceKIT there are two Xcore tiles (each has 8 cores). The below pie chart in figure 4.10: includes both tiles. Total memory is 128 kilo bytes (64 kilo bytes per processor).

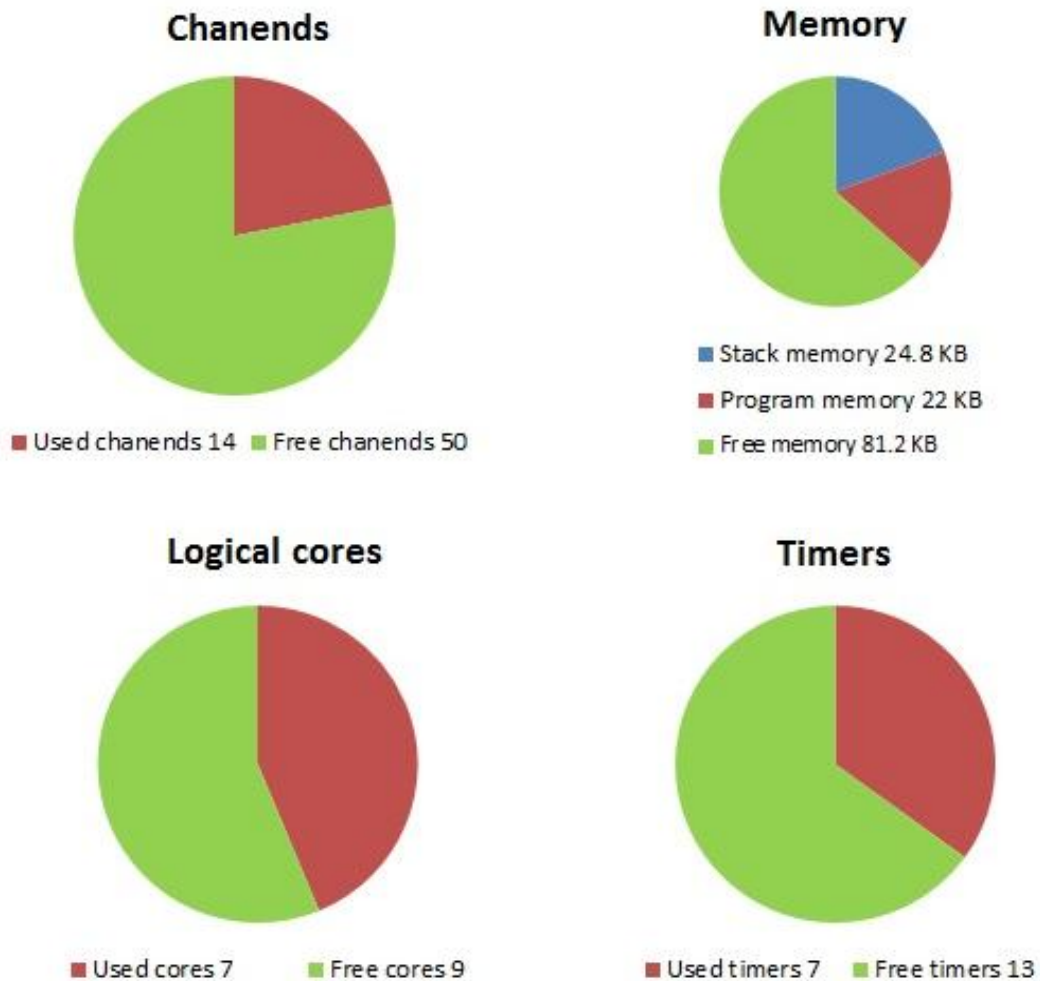


Figure 4.1: Resource utilization

4.5 Timing Results

FUNCTION	TIME
Frame Subtraction + Binarisation	38 ms
CCA	117 ms
Annotation	4 ms

Table 4.3: Timing results

SUMMARY

1. Memory analysis is important in embedded systems in order to design an application with lowest memory consumption.
2. When we build a XC code, the compiler converts it to a binary file. This binary file is loaded into XMOS processor to run the application in XMOS hardware
3. In sliceKIT there are two Xcore tiles (each has 8 cores), total memory is 128 kilo bytes (64 kilo bytes per processor).
4. From timing result we can observe that connected component analysis (CCA) stage takes more time.

CHAPTER 5

PERFORMANCE IMPROVEMENT

5.1 Parallel CCA

The parallel algorithm uses a divide and conquer approach. This algorithm divides the image, and then obtains the features of the objects in each sub-image. The features are passed from right to left; hence a sub-image to the left will have a higher precedence than one to the right. The algorithm has four major steps as shown in Figure 5.1.

In the first step, the image is divided into K sub-images using vertical cuts of the image where the last column of each sub-image overlaps with the first column of its successive sub image. K can be any value greater than one. For the purpose of explanation it is assumed to be 4. Each sub-image undergoes a single pass connected component analysis labeling procedure. At the end of the analysis, each sub- image will have for itself a corresponding merger table (implemented in an array) along with arrays to hold the features of components that are present in each sub-image. Also a border table is maintained which stores the labels assigned at the overlapping column. In other words, the labels of the last column and first column of all sub images are stored in border tables. They are in fact the same column just being assigned with different labels. The merger tables obtained can be visualized as graphs where the nodes are the labels and an interconnection between nodes implies the labels are equivalent. In this algorithm, equivalences can also be implied by observing the border table of a neighboring partition. For example, consider the mirror image of a C shaped object that is cut in the middle. The partition in which the two ends of the object are present will be counted as two objects. This problem is resolved by resolving the merger table of a given sub-image using the information in its border table and the labels present in the border table of the partition neighboring it.

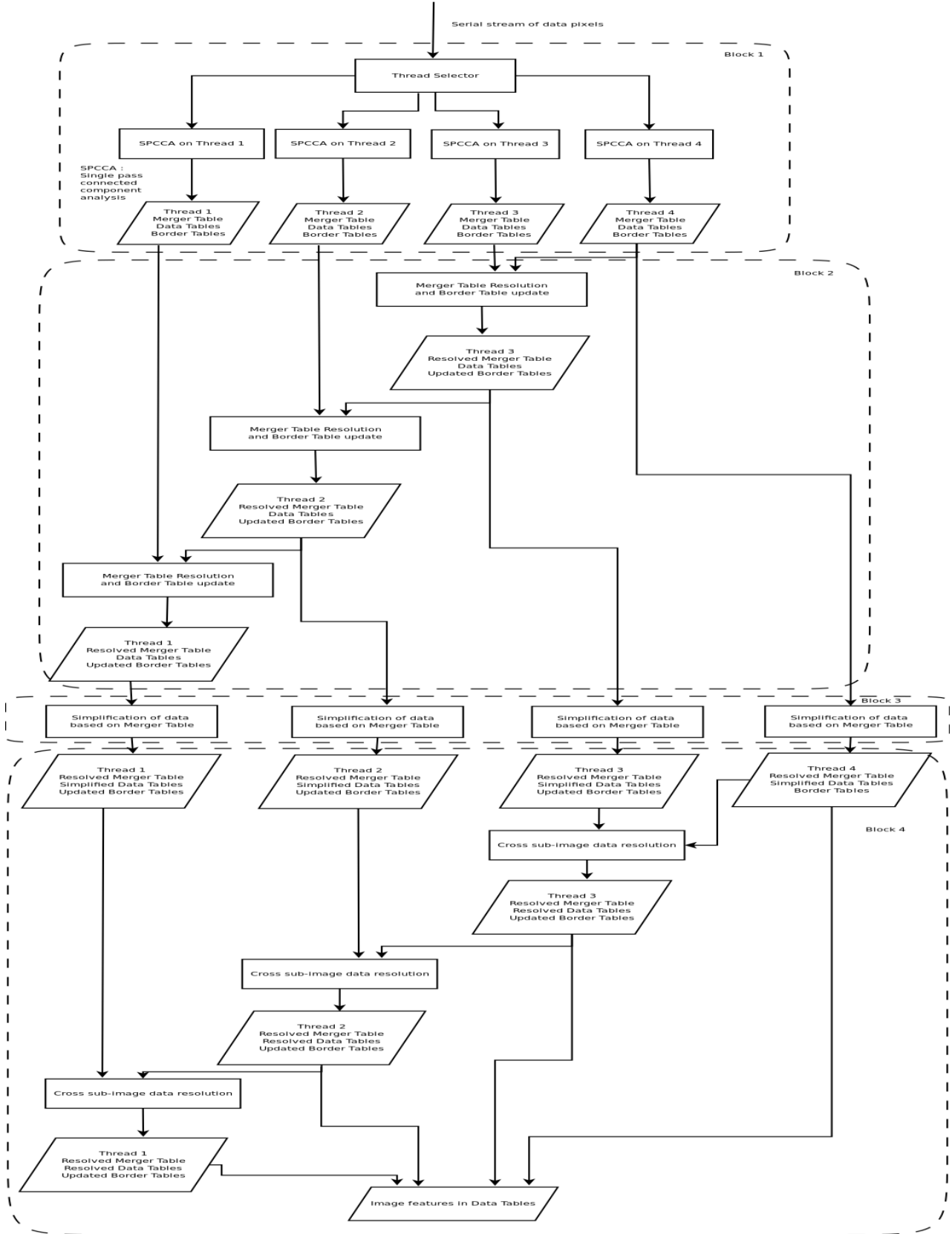


Figure 5.1: Block diagram of proposed parallel CCA

Consider Figure 5.2. When the features are passed from C to B, both the labels 1B and 2B will be possible candidates. To resolve this problem, the merger table must be resolved to show that 1B and 2B are equivalent labels. In this case, the merger table of sub-image B is resolved using the left border table of sub-image C. Although different labels (labels 1B and 2B) have been assigned in B for the object pixels on the border, the left border table of sub-image C contains an equivalent label for them. Hence equivalence between labels 1B and 2B is created. In the second case as shown Figure 5.3, the C shaped object is cut in the middle. Hence we obtain an analogous case to the previous one but in this case while the features are passed from C to B, the features of label 1C and label 2C are passed to the same label (label 1B) in sub-image B. As a result, in this case we need not resolve the merger table of C. Effectively, what is implied by these two cases is that it is sufficient to resolve the merger table of a given sub-image based on the information of the sub-image to its immediate right.

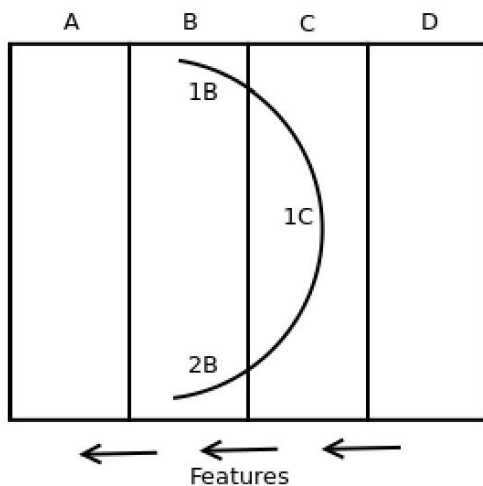


Figure 5.1: Passing features case I

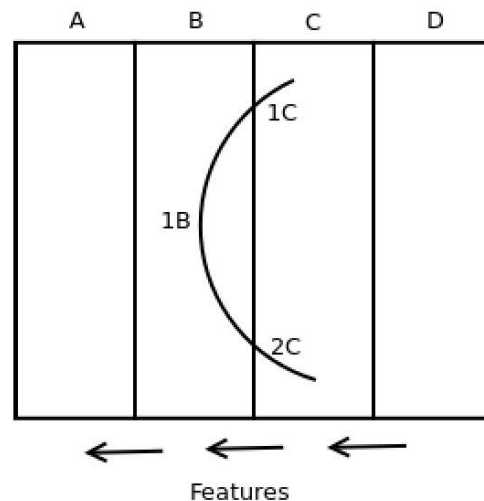


Figure 5.2 : Passing features case II

Hence, this step (Block 2 of Figure 5.1) involves simplifying the merger table of each sub-image by considering its right border table. It is sufficient to consider only the right border table as equivalences need to be derived only from the information contained in its right neighbour as explained. Hence we take the right border table of the sub-image and left border table of its right neighbour to resolve equivalences.

The next two steps in Figure 5.1 (Block 3 and Block 4) are straightforward. Data corresponding to features such as area and center of gravity of each node are pushed to the root of the node and the data present in the node is removed. In the case of area, the data which is number of pixels from a node is added to the root of the node and the area data of the node is made zero. This ensures that only root node possesses data.

The final step is to merge the information obtained from each sub-image. A sub-image on the left will have higher precedence than one on the right. We iterate through each non-zero label on the right border tables and add to the root of that label node, the data corresponding to the label at the same position in the neighboring sub-image's left border table. This implies that there is a component common to both A and B. Its features from B will be pushed to the component in A and its corresponding data in B will be removed ensuring we do not count the same component again. Finally, the total number of components in the entire image is the number of components with non-zero features in the sub-images.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this chapter, we give the summary of the work done and discuss possible future work.

6.1 Conclusion

The developed object tracking system uses 6 threads simultaneously while 1 thread is used initially to load the images into SDRAM in RGB565 format. Now it is possible to detect a particular object in each and every frame of the video. In addition, real time performance is another constraint. It has obtained a processing speed of 8 frames per second. So, it is required to increase processing speed for real time performance. Currently we are detecting all the moving objects in the video and put a bounding box around the objects with area threshold of 50 pixels.

6.2 Future work

Possible extensions are

1. From timing analysis it is clear that CCA is taking more time to label binary image. To increase processing speed, it is needed to optimize CCA by splitting it in to two parallel tasks and run in two different cores simultaneously.
2. Binary images will have more noise if large movements in the background such as moving clouds, leaves and ripples in the pond etc. We need to take care of this by adding appropriate filter.
3. Using a camera slice with an addition of one more thread we can make a more sophisticated system to track moving objects from live camera.

REFERENCES

- [1] D. G. Bailey, C. T. Johnston, “Single Pass Connected Components Analysis”, *Proceedings of Image and Vision Computing* pp. 282–287, *New Zealand, December 2007*.
- [2] PL. Siddharth , N. Sudha, Dan Wilkinson, “A Parallel Algorithm for Single-Pass Connected Component Analysis and its Realization on a Multi-Core XMOS Processor”
- [3] A. Yilmaz, O. Javed and M. Shah, “Object tracking: A survey,” *ACM Computing Surveys* (CSUR), Vol. 38, No. 4, 2006.
- [4] U. Ali, M. B. Malik and K. Munawar, “FPGA/Soft-processor based real-time object tracking system”, *Proc. IEEE Southern Conference on Programmable Logic* ,33-37, 2009.
- [5] H.M.Alnuweiti and V.K. Prasanna, "Parallel architectures and algorithms for image component labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(10), 1014-1034 (1992).
- [6] Jablonski, M., Gorgon, and M., "Handel-C implementation of classical component labelling algorithm", in *Euromicro Symposium on Digital System Design (DSD 2004)*, *Rennes, France*, 387-393 (2004).

[7] Dalal, N. and B. Triggs, “Histograms of oriented gradients for human detection”. In *Computer Vision and Pattern Recognition*, 2005. *CVPR 2005. IEEE Computer Society Conference on*, volume 1. 2005. ISSN 1063-6919.

[8] Yang, T., Q. Pan, J. Li, and S. Li, “Real-time multiple objects tracking with occlusion handling in dynamic scenes”. In *Computer Vision and Pattern Recognition*, *CVPR 2005. IEEE Computer Society Conference on*, volume 1. 2005. ISSN 1063-6919.

[9] Gonzalez, R. E. W., Rafael C. and S. L. Eddins, “Digital image processing using MATLAB”. *Pearson Education India, Reading, MA*, 2004.

[10] Jason, A. S. (). Controlling external SDRAM. Last access May 5th 2014. URL https://github.com/xcore/sc_sdram_burst.

[11] XMOS(). xtimecomposer user guide, Last access May 5th 2014. URL <https://www.xmos.com/download/public/xTIMEcomposer-User-Guide%28X3766B%29.pdf>

[12] XMOS(). slicekit hardware manual, Last access May 5th 2014. URL <https://www.xmos.com/support/documentation/xkits?subcategory=sliceKIT&product=15826&component=16091>

[13] XMOS(). Programming XC on XMOS Devices , Last access May 5th 2014. URL [http://www.xmos.com/download/public/Programming-XC-on-XMOS-Devices\(X9577A\).pdf](http://www.xmos.com/download/public/Programming-XC-on-XMOS-Devices(X9577A).pdf)

14. Shah, M. (2012). Histogram of oriented gradients (HOG), Last access May 5th 2014. URL <http://www.youtube.com/watch?v=0Zib1YEE4LU>