

**ADDRESSING EFFICIENCY ISSUES
OF
VIDEO SURVEILLANCE ALGORITHMS**

A DISSERTATION

*Submitted in partial fulfillment of the
requirements for the award of the degree
of*
**MASTER OF TECHNOLOGY
in
COMPUTER SCIENCE AND ENGINEERING**

By

KSHITIZ GUPTA



**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE -247 667 (INDIA)
JUNE, 2009**

Candidate's Declaration

I hereby declare that the work being presented in the dissertation report titled "Addressing efficiency issues of video surveillance algorithms" in partial fulfillment of the requirement for the award of the degree of Master of Technology in Computer Science and Engineering, submitted to the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, is an authentic record of my own work carried out under the guidance of Dr. Ankush Mittal, Associate Professor and Dr. Manoj Mishra, Professor in Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee. I have not submitted the matter embodied in this dissertation report for the award of any other degree.

Dated: 15th JUNE 2009

Place: IIT Roorkee



(Kshitiz Gupta)

Certificate

This is to certify that above statements made by the candidate are correct to the best of my knowledge and belief.

Dated: 15th JUNE 2009

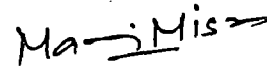
Place: IIT Roorkee.



Dr. Ankush Mittal,

Associate Professor,

Department of Electronics
and Computer Engineering,
IIT Roorkee, Roorkee,
247667 (India).



Dr. Manoj Mishra,

Professor,

Department of Electronics
and Computer Engineering,
IIT Roorkee, Roorkee,
247667 (India).

ACKNOWLEDGEMENTS

I am thankful to Indian Institute of Technology Roorkee for giving me this opportunity. It is my privilege to express thanks and my profound gratitude to my mentor Dr. Ankush Mittal, Associate Professor for his invaluable guidance and constant encouragement throughout the dissertation. I was able to complete this dissertation in time due to the constant motivation and support received from him.

I am also grateful to Dr. Manoj Mishra, Professor to allow me to independently explore my ideas by not allowing any necessary disturbances from administrative and academic work.

I am also grateful to Dr. Praveen Verma, Postdoctoral Researcher in Department of Computer Science, University of Missouri at Columbia for helping me to understand some basic and important concepts explored in the dissertation work. I am grateful to Mr. Salil Shirish Sahasrabudhe, Mr. Khalil Sawant, Mr. Tarun Kumar, Mr. Nityam Parakh and Mr. Payas Goyal, my colleagues, for being excellent peers and creating a congenial environment for work. I am also thankful to all my friends who helped me directly and indirectly in completing this dissertation.

Most importantly, I would like to extend my deepest appreciation to my family and friends for their love, encouragement and moral support.



(Kshitiz Gupta)

ABSTRACT

Video surveillance systems have found widespread use in applications ranging from homeland security for human lives and property, traffic monitoring, law enforcement practices and military defense. The scientific challenge is to devise and implement automatic systems able to detect and track moving objects, and interpret their activities and behaviors. The other issues that have come up are in terms of efficiency of the algorithms involved in video surveillance. Issues like robustness, scalability and real time processing are hindering the progress of video surveillance.

The onset of affordable multiprocessors has triggered a shift from clusters to programming using multiprocessors. There is a great enthusiasm in the industry as well as the academic community about the change parallel programming is bringing in. The idea with parallel programming is to use the massive performance given by these multiprocessors to solve the algorithms of video surveillance in real time. Alternatively they can be used to work on video streams of better resolution as well.

We have implemented video surveillance algorithms in a way to reduce the amount of time that is taken to process one frame. The implementations include a intuitive fusion algorithm. We have shown the applications of support vector machines to solve background modeling. We have also implemented background modeling, Embedded Zero Tree Wavelet algorithm, morphological operations and connected components labeling on the GPU and achieved considerable speed up. The idea was to find out the applicability of the GPUs in the field of video surveillance.

Table of Contents

| | |
|------------------------------------------------------|-----|
| Candidate's declaration..... | i |
| Acknowledgements..... | ii |
| Abstract..... | iii |
| Contents..... | iv |
| List of figures..... | vii |
| List of tables..... | ix |
| | |
| 1. Introduction | 1 |
| 1.1 Introduction | 1 |
| 1.2 Motivation | 2 |
| 1.3 Problem Statement | 2 |
| 1.4 Organization of Report | 3 |
| | |
| 2. Fusion Algorithm | 4 |
| 2.1 Background Modeling | 4 |
| 2.2 Gaussian Distribution | 6 |
| 2.3 Using Gaussians for background modeling | 7 |
| 2.4 From Single Gaussian to Multiple Gaussians | 8 |
| 2.5 Motivation for fusion algorithm | 10 |
| 2.6 Design and Implementation | 10 |
| 2.7 Experimental Results | 12 |
| 2.8 Conclusions | 14 |
| | |
| 3. Background Modeling using Support Vector Machines | 16 |
| 3.1 Introduction to support vector machines | 16 |
| 3.2 One class classification | 16 |

| | |
|--------------------------------------------------------------|----|
| 3.3 Errors in one class classification | 17 |
| 3.4 Introduction to data description toolbox | 18 |
| 3.5 Applying one class classification to background modeling | 18 |
| 3.6 Results and observations | 20 |
| | |
| 4. Parallelization of EZW on CUDA architecture | 23 |
| 4.1 Introduction | 23 |
| 4.2 nVidia CUDA architecture | 23 |
| 4.2.1 Programming Model | 25 |
| 4.2.1.1 Thread Hierarchy | 25 |
| 4.2.1.2 Memory Hierarchy | 26 |
| 4.2.2 GPU Implementation | 27 |
| 4.3 EZW algorithm | 28 |
| 4.4 Our Algorithm | 32 |
| 4.5 Results | 35 |
| 4.6 Conclusions | 37 |
| | |
| 5 Background modeling on MultiCore Processors | 38 |
| 5.1 Background modeling using single Gaussian | 38 |
| 5.2 STI Cell BE | 40 |
| 5.2.1 Hardware Architecture | 40 |
| 5.2.1.1 PowerPC Processor Element | 40 |
| 5.2.1.2 Synergistic Processor Elements | 41 |
| 5.2.1.3 Element Interconnect Bus | 41 |
| 5.2.2 Software Development Kit | 42 |
| 5.3 Parallelization on Cell BE | 42 |
| 5.3.1 Work done on PPE | 42 |
| 5.3.2 Work done on SPE | 43 |
| 5.4 Parallelization on GPU | 44 |
| 5.5 Results and conclusions | 46 |

| | |
|-------------------------------------------------------------------------------|-----------|
| 6 Parallel Blob Segmentation | 47 |
| 6.1 Introduction | 47 |
| 6.2 Background modeling and detection of foreground and background regions | 48 |
| 6.3 Binary morphological operations | 50 |
| 6.4 Connected Components Labeling | 52 |
| | |
| 7 Conclusions and Future Work | 59 |
| | |
| References | 61 |

List of Figures

| | |
|-------------------------------------------------------------------------------------------------------|----|
| Fig.2.1 Image and its residual after background subtraction..... | 4 |
| Fig 2.2 Gaussian distribution with different values of mean and standard deviation..... | 7 |
| Fig 2.3 Fusion Algorithm..... | 11 |
| Fig 3.1 Images used as training data | 20 |
| Fig 3.2 Images showing the output of background modeling using SVM..... | 21 |
| Fig 4.1 Figure showing number of floating point operations for CPU and the GPU.... | 24 |
| Fig 4.2 Arrangement of threads..... | 25 |
| Fig 4.3 Figure showing how threads access global, shared and local memory | 26 |
| Fig 4.4 Parent Child Dependencies of Subbands of wavelet coefficients | 28 |
| Fig 4.5 Morton Scan order on a matrix of wavelet coefficients at three level decomposition | 29 |
| Fig 4.6 EZW Algorithm | 30 |
| Fig 4.7 Wavelet coefficients of an 8 x 8 image..... | 31 |
| Fig 4.8 Figure showing the trees built for the horizontal, vertical and diagonal details .. | 32 |
| Fig 4.9 Comparison of running times of parallel implementation against linear implementation | 36 |
| Fig 4.10 Speed up of parallel implementation over linear implementation with varying image sizes..... | 37 |
| Fig 5.1 Flowchart for background modeling | 38 |
| Fig 5.2 Parallelization of background modeling on CBE | 43 |
| Fig 5.3 Input and corresponding background modeled images | 46 |
| Fig 6.1 Application of GMM, morphological operations and CCL algorithms | 48 |
| Fig 6.2 Tile structure on GPU where every pixel is processed by one thread | 49 |
| Fig 6.3 A 3 x 3 structuring element | 51 |
| Fig 6.4 Figure shows the race conditions on boundary pixels | 52 |
| Fig 6.5 Output after applying CCL to an image | 53 |
| Fig 6.6 Figure showing the 4 nbrs for pixel p | 54 |
| Fig 6.7 Figure showing the pixels that need to checked for overlap..... | 55 |
| Fig 6.8 Algorithm for resolving two neighboring regions | 56 |

List of Tables

| | | |
|-----------|-----------------------------------------------------------------------------------------------------------------------------------------|----|
| Table 2.1 | Table showing the number of small objects and the time required to model frames when Mixture of Gaussians algorithm is used..... | 13 |
| Table 2.2 | Table showing the number of small objects and the time required to model frames when Fusion algorithm is used..... | 14 |
| Table 3.1 | Types of errors in one class classification problem | 18 |
| Table 3.2 | Table showing the training data for a 5 x 5 block of image..... | 19 |
| Table 3.3 | Running times for background modeling for three approaches..... | 21 |
| Table 4.1 | Table showing the relative performance of parallel algorithm against a standard linear implementation..... | 36 |
| Table 4.2 | Table showing the corresponding speedups at different image sizes..... | 36 |
| Table 5.1 | Table showing the running times of background modeling for three implementations | 46 |
| Table 6.1 | Table showing the running time of GMM on image size of 320 x 240 pixels for sequential and parallel implementation..... | 49 |
| Table 6.2 | Table showing the running time of one morphological operation on an image of 320 x 240 pixels with different structuring elements | 52 |
| Table 6.3 | Running times of CCL on linear and parallel implementation for images of varying sizes | 56 |

1.1 Introduction

Security of human lives and property has always been a major concern for civilization from several centuries. There is a growing need for improved safety and security against the ever increasing threats of theft, accidents, terrorists' attacks, riots and natural calamity. For negotiating these increasing threats video surveillance systems are finding wide spread usage. Video surveillance systems have been deployed in various areas for providing homeland security, traffic monitoring etc. In April 2009 India launched RISAT 2 to provide surveillance along the Indian borders. The 300 kilogram satellite shows any movement across the surface of earth from a height of 550 kilometers. It is used for monitoring the country's borders round the clock, checking cross-border movement and helping the Indian security forces in anti-infiltration or anti-terrorist operations.

Video surveillance systems have undergone a lot of changes from the days of simple analog Closed Circuit Television (CCTV) cameras to multimodal and distributed systems. Development of video surveillance systems has been helped by the reduction in prices of the cameras. Complex multimodal systems are being conceived because of the availability of affordable sensors. Another area that has driven growth of video surveillance systems is the growth in networking infrastructure. IP-based cameras have enabled efficient deployment of cameras at any remote site over the existing wired or wireless network without the requirement of bulky co-axial fibre cable connections and dedicated processors. These have resulted in deployment of large scale video surveillance systems with potentially thousands of cameras distributed over widespread geographical locations.

The scientific challenge is to devise and implement automatic systems that are able to detect and track moving objects, and interpret their activities and behaviours. The other issues that have come up are in terms of efficiency of the algorithms involved in video surveillance. Issues like robustness, scalability and real time processing are hindering the progress of video surveillance.

1.2 Motivation

Video surveillance is one of the fastest growing sectors in commercial market. It is an active area of research. The aim of automatic video surveillance is to automatically detect the interesting objects in the monitored area, track their motion and automatically take appropriate action like alerting a human supervisor. Second-generation surveillance systems constitute the current state of the art from a commercial viewpoint. The main technical innovation in second-generation surveillance systems is the introduction of digital video representation. Second-generation surveillance systems had first separately explored the advantages of digital approaches to acquisition, transmission, processing, storage, and visualization. With the recent advancements in video and network technology, there is a proliferation of inexpensive network based cameras and sensors for widespread deployment at any location, as well as with the development of new video-processing and computer-vision algorithms allowing more complex scenes to be considered. All this progress has made it possible and necessary to consider a new perspective in this field in order to exploit the advantages of a fully digital approach which finally led to the growth of third generation surveillance systems. Making video surveillance systems more robust and automated creates an opportunity to make use of newer and newer approaches. Nearly infinite variability of the environmental factors and the open-ended goals of many surveillance problems conspire to create situations where even the most advanced detection, tracking and recognition algorithms falter. Making video surveillance systems real time is now more possible than ever because of availability of affordable parallel computing systems. Large amounts of video data can be processed in parallel in these systems which can help to process more frames (of larger resolution) per second. Any video surveillance system performs background modeling as the first task. In background modeling the frame's pixels are segregated in foreground and background pixels. The performance of the video surveillance system depends on the background modeling task.

1.3 Problem Statement

In this dissertation work we propose and implement the background modeling module for video surveillance while addressing the real time processing aspect of the

problem. We aim to use parallel processing, support vector machines and a fusion algorithm to speed up background modeling. We focus on a compression algorithm Embedded Zero Tree Wavelet (EZW). We make use of parallel processing to speed up the execution of this algorithm to make it useful for real time transmission. We also focus on parallelizing morphological operations for binary images. The aim of morphological operations is to make the image smoother. Performing background modeling may leave some holes in the image; these holes are filled by using morphological operations. This image is then processed by connected components labelling logic. This code is also implemented in parallel. The idea behind connected components labelling is to group the foreground objects in logical objects.

1.4 Organization of the report

Chapter 2 gives the basics of background modeling explaining how Gaussians are used in background modeling. This chapter then proceeds to develop the fusion algorithm which answers the latency issue in an intuitive fashion.

Chapter 3 explains how support vector machines can be used to perform background modeling.

Chapter 4 gives a CUDA based implementation of the EZW algorithm explaining the tweaks done to parallelly solve the algorithm.

Chapter 5 gives a comparison of implementation of background modeling tasks on Cell BE and that using nVidia CUDA architecture.

Chapter 6 gives a CUDA based implementation of parallel blob segmentation. This chapter covers three implementations Gaussian mixture models, binary morphological operations and Connected Components Labelling

Chapter 7 presents the conclusions of the work and suggests future work that can be done to extend the work.

2.1 Background Modeling

Background modeling can be understood as the procedure by which the background of a video file is modelled against changes. The aim is to separate out the foreground and background pixels. Background modeling is a very basic step in the process of video surveillance.

Many computer vision and image processing systems, particularly those in areas such as surveillance, human-computer interaction, and 3D model reconstruction, rely heavily on an early step, commonly called “background subtraction” that segments the scene into foreground and background regions [1]. Background subtraction approach is based on obtaining a background or reference model which is then subtracted with the current image. Background subtraction is quite adept at extracting object information but it is sensitive to dynamic changes in the environment and the background model needs to be dynamically updated. Other way of getting objects’ information is to use inter-frame differences. This does not have any issues like that of being getting updated. However the objects are only seen by their edges.

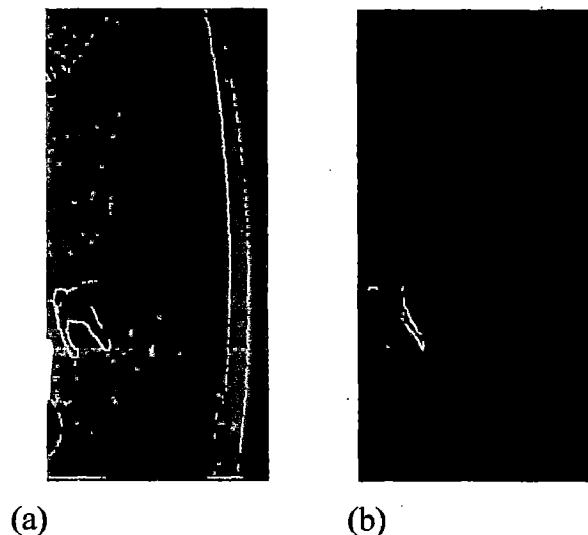


Figure 2.1

(a) Original Image (b) Image of the residual got after background subtraction

Background modeling assumes that the video scene is composed of a relatively static model of the background, which becomes partially occluded by objects that enter the scene. These objects (usually people or vehicles) are assumed to have features that differ significantly from those of the background model (like their colour or edge features). The terms foreground and background are not scientifically defined however and thus their meaning may vary across applications. For example, a moving car should usually be considered as a foreground object but when it parks and remains still for a long period of time, it is expected to become background. Also, not all moving objects can be considered foreground. The simplest approach is to record an image when no objects are present and use this image as the background model. However, continuous updating of the model is required to make the foreground extraction more robust to the gradual changes in lighting and movement of static objects that are to be expected in outdoor scenes. A robust system should not depend on careful placement of cameras [2]. Background modeling finds numerous applications and is a module of every video surveillance task. It finds application in object tracking where it can be used to find a particular object in a given frame. Background modeling is used in security where it can be used to identify foreground objects and activities and then alert the authorities in case of abnormal event. It will be an important part of the futuristic cars that shall drive on their own by figuring out what object lay ahead in the path.

Gaussian distributions have been used for modeling background in [3] and [4]. Francois et. al [3] have preferred to use HSV (Hue Saturation Value) colour space over RGB (Red Green Blue) colour space. They point that the dissimilar information of HSV colour space axes offers a more intelligent way of modeling background. Stauffer et. al [4] have used multiple Gaussians to improve the robustness of background modeling in situations where the background is multimodal. [5] presents an interesting discussion on the type of issues that can come up when performing background modeling.

As an alternate to just use colour information various approaches have been discussed in which edge information has been used. Jabri et. al. [6] fused the information of edges as well as colour of pixels. They used sobel operator for edge detection. However the usage of colour information presented with an issue if the colour

changed too drastically. Jain et. al. [7] suggested using of sub pixels to capture very small translations of objects. Sub pixels are interpolations of existing pixels which are also used in video compression [8]. [9] shows how k-means clustering can be used for background modeling. They used two clusters, one for background pixels and other for foreground pixels.

2.2 Gaussian Distribution

We now give a brief description of the Gaussian distribution. Mixture of Gaussians has been one of the most widely used mathematical tools for background modeling. The Gaussian distribution may be defined by two parameters, location and scale: the mean ("average", μ) and variance (standard deviation squared) σ^2 .

The importance of the normal distribution as a model of quantitative phenomena in the natural and behavioural sciences is due in part to the central limit theorem. Central limit theorem states that the sum of a large number of independent and identically-distributed random variables will be approximately normally distributed (i.e., following a Gaussian distribution, or bell-shaped curve) if the random variables have a finite variance. Many measurements, ranging from psychological to physical phenomena can be approximated by the normal distribution. While the mechanisms underlying these phenomena are often unknown, the use of the normal model can be theoretically justified by assuming that many small, independent effects are additively contributing to each observation. We can think of the Gaussian distribution as the most "general" distribution when anything comes to model a natural physical phenomenon.

2.3 Using Gaussians for background modeling

Every background pixel is modelled by using a Gaussian distribution. The distribution has its own mean and variance. A background model for every pixel is stored from some of the initial frames. If a frame is of 100 x 100 size then we will have 10000 pixels and each of these pixels will have 3 components red, green and blue. So for storing the background model for this frame we need to store 30000 Gaussians. Each of these Gaussians will have its mean and variance.

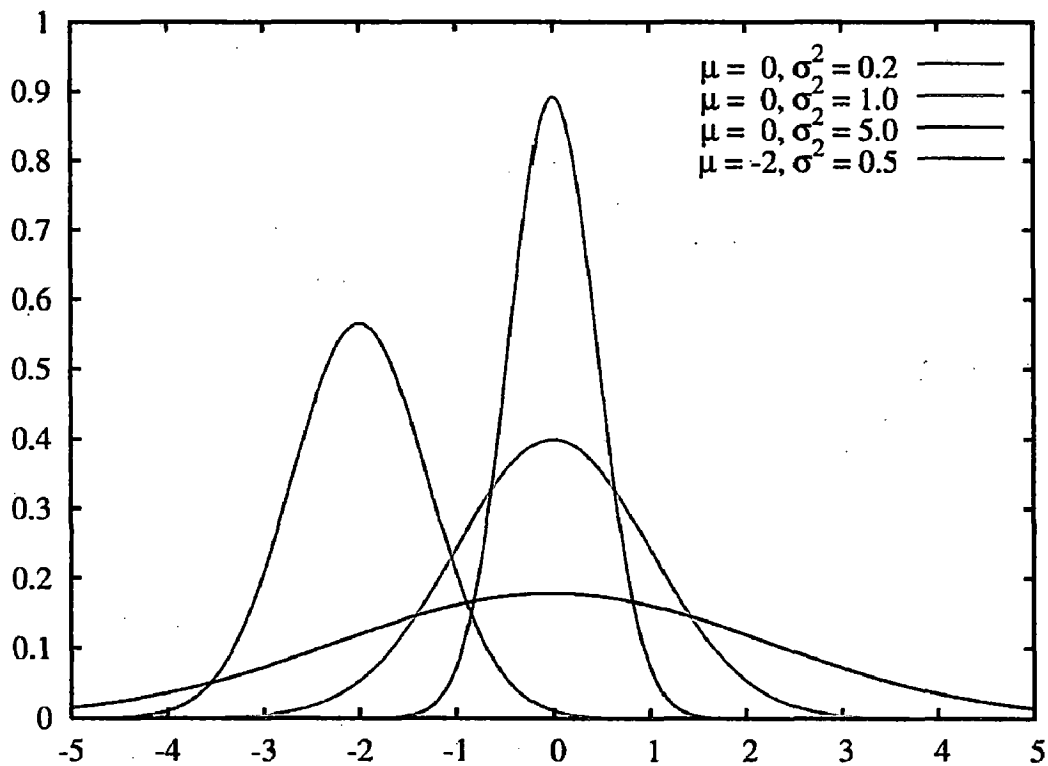


Figure 2.2: Gaussian distribution with different values of mean and standard deviation [7]

Every pixel of every new frame is compared against the model distribution. The decision of whether the pixel is foreground or background is made on the basis of a threshold. The value of pixel is checked if it lies in between 2.5 standard deviations of mean of the model stored. If the value lies in between this range then it is considered as a background pixel and a pixel that matched the background model. Otherwise the pixel is considered as a foreground pixel.

The background model for each pixel is then updated according to the pixel value in the newer frame. If μ is the mean of the background model and σ is the standard deviation and x be the pixel under consideration of the incoming frame.

$$\mu \leftarrow (1 - \alpha)\mu + \alpha x$$

$$\sigma^2 \leftarrow \max(\sigma_{\min}^2, (1 - \alpha)\sigma^2 + \alpha(x - \mu)^2)$$

Here α is the learning rate. More the value of α quicker the system updates the background model with the incoming frames. σ_{\min} is a value that is forced by the system so to ensure that the value of standard deviation does not become too small to accept background pixels.

2.4 From Single Gaussian to Multiple Gaussians

The drawback of using single Gaussian is that it cannot work well if the background contains a dynamic object. If every pixel was a result of a single surface under fixed lighting then this could be described by just one Gaussian. But the case is rarely so. In real life conditions the lighting changes and this requires the Gaussian distribution to be adaptive. This means that the mean and the variance of the Gaussian need to be updated to reflect the change in lighting. A further complexity comes about because a particular pixel could be a result of more than one surface. We can have the same pixel to be there because of the motion of a car when the car is moving across and then result to be something else when the car moves. So we need more than one Gaussian to model the background. Incorporating both the complexities we need more than one adaptive Gaussian per pixel.

Let us consider an example to explain the need of multiple Gaussians. Consider a tree with moving leaves. Further consider one pixel of that region where this moving tree is. Now this particular pixel can be lighted by either the green colour of the leaf of the tree or by the blue colour of the sky. We would want both the surfaces to be considered as background. So for this we need two Gaussians. One of the Gaussians will accept the green colour of the leaf as background and other Gaussian will be accepting the blue colour of the sky as background.

Now if some red coloured car becomes an occluding object, both the Gaussians will not accept this pixel as background and the pixel will become marked as a foreground pixel.

Initialisation of the Gaussians is done in the same way as it was done for the case of single Gaussian. However, now the Gaussians also store a weight. Higher weight

indicates that the Gaussian is more important to the model. Let us assume that we are storing 3 Gaussians per pixel. So for an image of 100 x 100 pixels, we have 30000 Gaussians. Each Gaussian now stores a mean, standard deviation and a weight. For every pixel of the newer frame now, this particular pixel is checked if it lies within 2.5 standard deviations of the mean of every Gaussian of the model. If any of these Gaussians accepts this pixel as background then the pixel is marked as a background pixel. Each Gaussian also remembers if the pixel matched it or not. The weights of the Gaussians that matched the pixel are updated to increase their significance in the model.

$$\omega_{k,t} = (1 - \alpha)\omega_{k,t-1} + \alpha(M_{k,t})$$

Here $\omega_{k,t}$ is the weight of the k^{th} Gaussian for t^{th} pixel. $M_{k,t}$ is a variable that is either 1 or 0 depending on whether the Gaussian matched the pixel or not.

If none of the distributions matched the pixel in question then the least matching distribution is replaced to incorporate this new pixel. The new distribution has the current pixel value as its mean and a very high variance. A high variance is used so to ensure that anything even remotely related to this pixel in future is caught as background.

The mean and deviations for the distribution that did not match are not changed. The mean and deviations for the distributions that matched were however updated. The idea behind this is that a new pixel does not destroy the unmatched distributions. They are removed only when they become the least significant. The weight however for these distributions keeps on going down. This works in the case where one object enters the frame for just enough time to become part of the background and then starts to leave the frame. To remedy this situation the previous background distribution are present with the same mean and deviation to take over. A per pixel/ per distribution threshold allows freedom to work with different parts of the frame.

2.5 Motivation for Fusion Algorithm

High computation power and/or memory requirement is/are an important issue in background modeling. Using a single Gaussian takes considerably less time to model background than that taken by multiple Gaussians. Using a Single Gaussian implies lesser comparisons and fewer updations and a single Gaussian ends up being faster than Mixture of Gaussians. However, multiple Gaussians algorithm is more robust to repeated motions such as moving of a tree in wind or that of a flag. But when it comes to model background, when the background is static then we see that the performance of finding interesting pixels of a single Gaussian is comparable to that when Mixture of Gaussians is used. This interesting observation makes suggestion to fuse both the algorithms to exploit this trade-off.

Consider a windy day when we are modeling the background for a moving tree. In the presence of the wind, the swaying tree can cause a false detection as a foreground object, so the logical choice is to use a mixture of Gaussian which will suppress these false detections. But when the wind velocity reduces and the tree ceases to move then it can be modelled well by only a single Gaussian, as it is now a static object.

The work presented here is most similar to the work done by Shimada et. al [11]. They varied the number of Gaussians that are used to model a pixel depending on the nature of illumination changes that occur at that pixel. To speed up background modeling Hyo-kak et. al [12] segregated the pixels and used a near search technique. Their idea was based on temporal persistence, spatial compactness and spatial translation. They reduced the number of pixels on the actual computation needed to be performed.

2.6 Design and Implementation

The fusion algorithm described in the Figure 2.3, models background for some of the frames by using just one Gaussian and for other frames by using a mixture of Gaussians. The fusion algorithm consists of 3 aspects. For a window of M frames some frames are modelled using a single Gaussian others by mixture of Gaussians. And this is followed by switching logic.

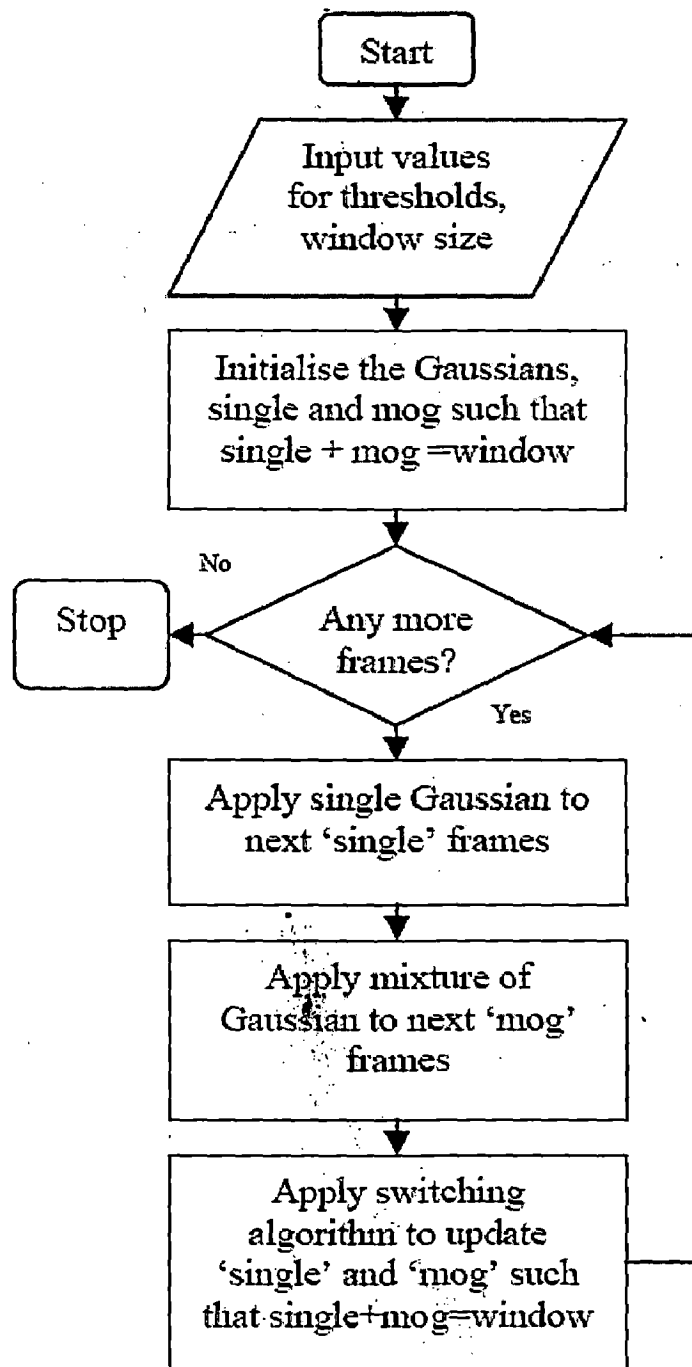


Figure 2.3: Fusion Algorithm

To make the right choice about whether to use single Gaussian or to use a Mixture of Gaussians (MOG) the algorithm runs a logic after a window of every M frames. Amongst these M frames some of the frames will be modelled by single Gaussian and other by

MOG depending on the type of environment found. After these M frames the connected components algorithm is applied to the last frame. The objects detected are compared against a threshold for the size of the object. All the objects smaller than this threshold are counted. Then the number of such objects is compared against another threshold. If the number of small objects is more than a certain number this indicates the presence of a dynamic environment. Following this the number of frames for which MOG is supposed to run is increased. If the number of objects detected is less than the threshold then the number of frames for which single Gaussian is run is increased.

In a static environment the MOG algorithm will run just once per the window size. This may cause the MOG values to not stay updated with the current image pixels. For this the values of the single Gaussian are used to update the Gaussian which has the minimum weight from K Gaussians. It should be noticed that these weights are maintained as part of the MOG implementation so their updation does not add any overhead.

It is to be noticed that the code that runs for the switching logic should not become the bottle neck for the entire algorithm. If it so happens that the switching logic itself is very slow then the gains by running the single Gaussian for more frames will be lost. But as it turns out that the switching logic makes use of connected components algorithm and thresholding for the size of the small objects and connected components algorithm is already part of any standard background modeling module. So the switching logic even though it may seem as an overhead is not really an overhead.

2.7 Experimental Results

We have used a set of images as input data that has the leaves of the tree moving for a certain amount of frames which are followed by some frames when the wind stops. The first part of image set points towards the dynamic environment and the second part of the image set points towards the static background case.

The experimental results presented here are under the fair underlying assumption that MOG is a better approach to model backgrounds for video surveillance. We have modelled the same video first with only MOG and then with the fusion algorithm. It is found that both the algorithm give a relatively similar performance in the number of small objects but the fusion algorithm makes smart decision when the wind stops and starts working on more frames with single Gaussian algorithm greatly reducing the amount of time taken. As we have already mentioned that switching logic is really not an overhead so we discount its running time for our calculations.

| Frame Number | Time (s) | Number of Objects | Frame Number | Time (s) | Number of Objects |
|--------------|----------|-------------------|--------------|----------|-------------------|
| 36 | 6.453 | 1714 | 51 | 9.205 | 0 |
| 37 | 6.135 | 1892 | 52 | 8.246 | 0 |
| 38 | 6.176 | 1568 | 53 | 9.756 | 0 |
| 39 | 6.239 | 1699 | 54 | 7.278 | 0 |
| 40 | 6.344 | 1658 | 55 | 8.824 | 0 |
| 41 | 6.352 | 1613 | 56 | 7.413 | 0 |
| 42 | 6.376 | 1775 | 57 | 6.274 | 0 |
| 43 | 8.777 | 1579 | 58 | 6.898 | 0 |
| 44 | 9.165 | 1286 | 59 | 6.824 | 0 |
| 45 | 7.114 | 0 | 60 | 7.79 | 0 |
| 46 | 9.607 | 0 | 61 | 7.33 | 0 |
| 47 | 6.576 | 0 | 62 | 6.442 | 0 |
| 48 | 9.91 | 0 | 63 | 10.888 | 0 |
| 49 | 9.198 | 0 | 64 | 11.328 | 0 |
| 50 | 9.34 | 0 | 65 | 8.755 | 0 |
| | | | 66 | 9.533 | 0 |

Table 2.1: Table showing the number of small objects and the time required to model frames when MOG algorithm is used. Frames taken here correspond to that part of video where the environment shifts from dynamic nature to static nature.

Tables 2.1 and 2.2 show the relative performance of mixture of Gaussians against that of fusion algorithm. In case of fusion algorithm the values of number of objects are for every fifth frame because for our specific example we have taken a window size of 5 and we run the connected components algorithm once every 5 frames for our switching logic purpose. Switching time tells us the time that the code for making the decision took.

The point to be noticed here from the tables is that whenever switching logic was executed to make the choice the number of objects detected by the fusion stays same as it was in the corresponding case for pure MOG algorithm. As we can see from tables 2.1 and 2.2 frame number 40 has the same number of small objects for both the algorithms. This shows that we are not losing any object detection accuracy in the fusion algorithm.

| Frame Number | Algorithm | Time (s) | Number of Objects | Switching Time | Frame Number | Algorithm | Time (s) | Number of Objects | Switching Time |
|--------------|-----------|----------|-------------------|----------------|--------------|-----------|----------|-------------------|----------------|
| 36 | Single | 0.104 | | | 51 | Single | 0.118 | | |
| 37 | MOG | 6.21 | | | 52 | Single | 0.116 | | |
| 38 | MOG | 6.296 | | | 53 | Single | 0.113 | | |
| 39 | MOG | 6.71 | | | 54 | MOG | 6.726 | | |
| 40 | MOG | 8.935 | 1658 | 2.349 | 55 | MOG | 9.646 | 0 | 0.026 |
| 41 | Single | 0.184 | | | 56 | Single | 0.173 | | |
| 42 | MOG | 8.839 | | | 57 | Single | 0.204 | | |
| 43 | MOG | 8.949 | | | 58 | Single | 0.156 | | |
| 44 | MOG | 9.11 | | | 59 | Single | 0.167 | | |
| 45 | MOG | 7.692 | 0 | 4.213 | 60 | MOG | 7.921 | 0 | 0.019 |
| 46 | Single | 0.118 | | | 61 | Single | 0.113 | | |
| 47 | Single | 0.106 | | | 62 | Single | 0.102 | | |
| 48 | MOG | 6.858 | | | 63 | Single | 0.114 | | |
| 49 | MOG | 6.608 | | | 64 | Single | 0.104 | | |
| 50 | MOG | 6.675 | 0 | 0.037 | 65 | MOG | 6.527 | 0 | 0.054 |

Table 2.2: Table showing the number of small objects and the time required to model frames when Fusion algorithm is used. Frames taken here correspond to that part of video where the environment shifts from dynamic nature to static nature. One can see how the number of frames being modelled by single Gaussian increase as the environment becomes static towards the end.

2.8 Conclusions

The results have been obtained on a computer running on a Core2Duo 1.5 GHz processor with 2 GB RAM. The algorithms were coded in MATLAB 7. To test the

algorithm, we have used video sequences obtained from CAVIAR (Context Aware Vision using Imagebased Active Recognition) project site.

Fusion algorithm maintains a good balance between performance and latency. The intelligent switching ensures that we are using the right number of Gaussians depending on the environment. We see that fusion algorithm runs 80% of frames using mixture of Gaussians for a window size of 5 for cases that have dynamic backgrounds. This ensures that we have good performance for multi modal backgrounds. Latency issue can be better answered by increasing the window size. However, doing so we may start to see some performance deterioration as the updations will now be delayed.

Chapter 3 Background Modeling using SVMs

3.1 Introduction to SVMs

Support vector machines (SVMs) are a new method for classification of data. Support vector machines search for a linear hyperplane to separate data from two classes. In our case we have data corresponding to two classes only, foreground pixels and background pixels. By hyperplanes we simple mean planes of higher dimensions. SVMs take as input some training tuples and from this they try to find the optimal hyperplane. By optimal we mean that this particular hyperplane should have high accuracy in separating the classes.

The training time of SVMs can be extremely slow, but they are extremely accurate. They are also less prone to overfitting. The time required for classification is quite less which makes them useful to be used for background modeling in real time.

3.2 One Class Classification

We will first explain the motivation for selecting one class classification for solving background modeling. The problem of background modeling is to classify the pixels of a frame as background or foreground. Till this the problem is similar to any general classification. But in case of normal classification the training tuples that we have are belonging to classes, target class and outlier class. In case of background modeling the initial frames are all takes as background. So when we are creating the model we do not have any information about the class 'foreground' and thus normal classification cannot be used for background modeling.

The problem of one-class classification is a special type of classification problem. In one-class classification we are always dealing with a two-class classification problem, where each of the two classes has a special meaning. The two classes are called the target and the outlier class respectively:

Target class

This class is assumed to be sampled well, in the sense that of this class many (training) example objects are available. It does not necessarily mean that the sampling of the training set is done completely according to the target distribution found in practice. It might be that the user sampled the target class according to his/her idea of how representative these objects are. It is assumed though, that the training data reflect the area that the target data covers in the feature space.

Outlier class

This class can be sampled very sparsely, or can be totally absent. It might be that this class is very hard to measure, or it might be very expensive to do the measurements on these types of objects. In principle, a one-class classifier should be able to work, solely on the basis of target examples.

An example of one class classification could be the problem of machine diagnostics. Let the problem be to classify if the machine is running fine or faulty. It is simple to obtain measurements from a machine that is working. Please note that we are not indicating a situation where we want every possible measurement of a working machine. On the other hand getting a machine damaged in every possible way for training the model to learn about the faulty class is a very impractical and expensive approach. The consultant offering this solution has just lost his job.

Another example could be that of facial recognition for surveillance purposes. For this the target class is well defined but the outlier class could be anything. This would generate too many false detections if two class classification is used.

3.3 Errors in one class classification :

There are two types of errors in one class classification that need to be minimized. These two errors are called false positives and false negatives

The fraction false negative can be estimated by using cross validation on the target training set. In cross validation we make B batches of the training data set. Of these we use the B-1 batches for training of the SVM and use the last batch for testing the

| | | True Class Label | | |
|----------------|---------|-----------------------------------|--|------------------------------------|
| | | Target | | Outlier |
| Assigned Label | Target | True Positive Target Accepted | | False Positive Outlier Accepted |
| | | | | |
| | Outlier | False Negative Target Rejected | | True Negative Outlier Rejected |

Table 3.1: Types of errors in one class classification problem

SVM. This process is repeated over and over by changing the batches. The fraction false positive is however even more difficult to estimate if we do not have any outlier object available.

3.4 Introduction to data description toolbox

Data description (dd_tools) [13] is a MATLAB toolbox that provides tools, classifiers and evaluation functions for researching one class classification. The dd_tools is an extension of prtools. Prtools [14] is a MATLAB toolbox for pattern recognition. Dd_tools borrows and builds on the objects mapping and dataset provided by prtools.

For our implementation we made extensive use of the function svdd() which basically returns the model for a given data set. After the model is built this model is compared against the testing tuples. Based on this comparison a decision is made if the object belongs to target class or outlier class.

3.5 Applying one class classification to background modeling

The task at hand is to create a model of the area under surveillance and then use this model to separate the frame's pixels into background and foreground pixels. So the first task is to create the model. A model shall store data from a considerable number of frames. We use a number of frames to work around the acquisition noise of the

camera. If the same camera tries to capture an area twice the resulting images will not be identical because of the inherent acquisition of the camera. So we take more images of the area for building our model. This shall make the model more robust and results in lesser false detections. Creating of a model is very time consuming so we have two implementation choices on our hand. Either there could be a model for every pixel. If the frame is 100 x 100 in size then this means that we need to create 10000 models. The other choice is to create a block of say 5 x 5 pixels and have a model for each block. This reduces the number of blocks from 10000 to 400 which has a significant impact of the training time as well as the time required for classification. The downside is that we do lose some accuracy as we now get only 5 x 5 pixel blocks marked as foreground inside of having information for every pixel. But this information does not deter good tracking.

We now give a peek of what our training data looks like when we are creating a model. We used 34 frames to create the model. Each frame is 100 x 100 pixels in size and the block size for the model is 5 x 5 pixels. The training data is a 170 x 15 vector.

| | | Column Numbers | | |
|-------------|--------------------|------------------------|-------------------------|--------------------------|
| | | 1-5 | 6-10 | 11-15 |
| Row Numbers | For frames 1-34 | Data for pixel (1,1-5) | Data for pixel (1,6-10) | Data for pixel (1,11-15) |
| | For frames 35-68 | Data for pixel (2,1-5) | Data for pixel (2,6-10) | Data for pixel (2,11-15) |
| | For frames 69-102 | Data for pixel (3,1-5) | Data for pixel (3,6-10) | Data for pixel (3,11-15) |
| | For frames 93-136 | Data for pixel (4,1-5) | Data for pixel (4,6-10) | Data for pixel (4,11-15) |
| | For frames 127-170 | Data for pixel (5,1-5) | Data for pixel (5,6-10) | Data for pixel (5,11-15) |

Table 3.2: Table showing the training data for a 5 x 5 block of image

The model is a 20 x 20 array with one location holding the model for a 5 x 5 block of image.

Creation of a model is time taking process; however background modeling of future frames is very fast and robust. For background modeling we read the frames and then make 5 x 5 pixel blocks of the image in a similar way as we did when we created the

model. Dd_tools provides a '*' operator to find a comparison to the test data with the model.

3.6 Results and observations

We now provide the results that were obtained when performing background modeling using support vector machines. The results were obtained using MATLAB R2008a. The computer system was a Core2Duo 1.5 GHz with 2 GB RAM. The image set used was taken from [13].

The training data was a set of 34 images that had the area under surveillance without any occluding objects.

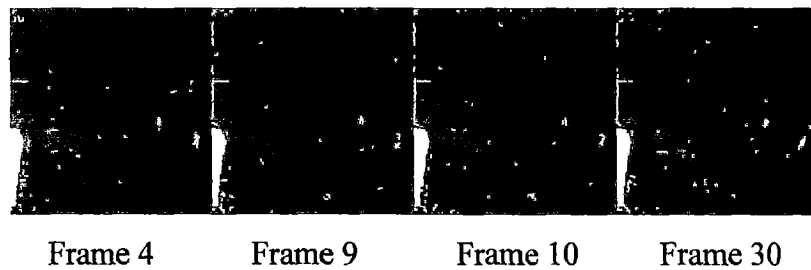
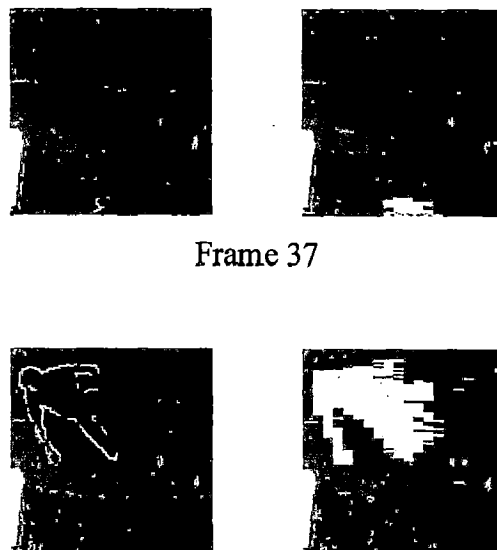


Figure 3.1: Images that were used as training data for the support vector machine

The following are the background modelled images that were obtained when SVM was used and data was processed in a 5 x 5 block of pixel



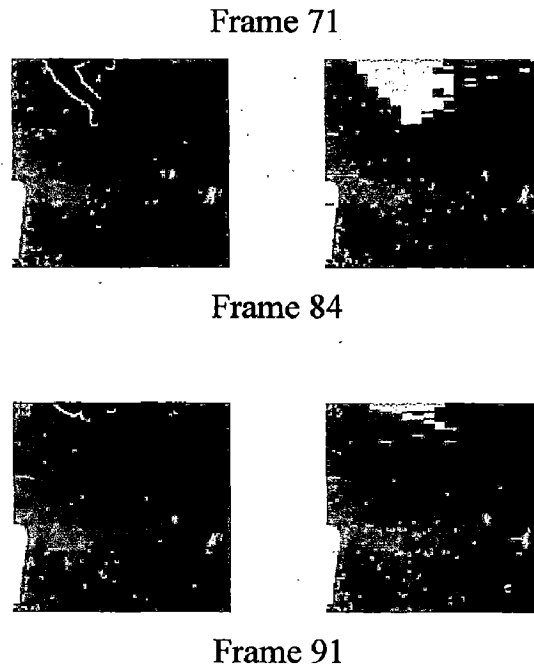


Figure 3.2: Images showing the output after a 5 x 5 block has been used for background modeling for SVMs.

The results obtained show the robustness of SVM for background modeling. There are very few outliers and since the outliers are of small size as compared to that of the actual object they can be suppressed easily by connected components algorithm. The following table gives the running times when background modeling is done for 66 frames using three approaches. The first approach uses a mixture of Gaussians that has three Gaussians. In the other approach we use one class classification and maintain a model for every pixel. In the third approach we maintain a model for every 5 x 5 pixels.

| | |
|---------------------------------------|------------------|
| Mixture of Gaussians with 3 Gaussians | 609.5709 seconds |
| SVM model per pixel | 388.7765 seconds |
| SVM model per 5x5 block | 12.7629 seconds |

Table 3.3: Running times of three approaches for performing background modeling for 66 frames

The general idea for the approach is that any occluding object shall be larger than a 5 x 5 pixel block. This approach will not give good results for outdoor surveillance as the environment can be dynamic with varying lighting condition and a dynamic

background. This approach can give good results for closed environments like ATM, cafeterias etc.

A good extension to this work could be a parallelization of the SVM training algorithm. The classification task is quite fast and would not require any parallelization. However, if the training time can be reduced then the performance of the over all system can be improved drastically. Updated SVM models will be able to take in the gradual changes that may happen for closed environments like change of lighting as day progresses.

Chapter 4 Parallelization of EZW on CUDA architecture

4.1 Introduction

The embedded zero tree wavelet (EZW) proposed by Shapiro [17] has a number of desirable properties. It has a superior compression efficiency which makes it suitable to be used for image and video compression. The algorithm works on the set of wavelet coefficients for the particular image to be compressed. Other aspect of EZW algorithm is its network resilience which makes it adept at handling situations that require real time transmission with low bandwidth. This makes the EZW algorithm suitable to be used for video surveillance. Another supporting reason for EZW to be used for video surveillance is that objects can be detected in the compressed domain, which can help avoid decoding of the EZW encoded bit stream [18].

Approaches to perform EZW in parallel have been proposed previously. In [19] Cheung et. al. discuss a parallel architecture for performing EZW compression. Ang et. al [20] give a good discussion about the merits and demerits of the scan order of the hierarchical trees of wavelet coefficients. They conclude that depth first search allows the complete encoding of one tree before proceeding on to the next tree. However their implementation caused a reduction in PSNR.

4.2 nVidia CUDA architecture

We now discuss a little about nVidia's CUDA (Code Unified Device Architecture) architecture which we have used to parallelize EZW. General purpose computing on the GPU is an active area of research. GPUs are already widespread. The performance of GPUs is improving at a rate faster than that of CPUs. The capabilities of the GPU have increased dramatically in the past few years and the current generation of GPUs has higher floating point performance than the most powerful (multicore) CPUs [16]. The GPU contains hundreds of cores that work great for parallel implementation. The programming is done in SIMD style where same code is worked on different data locations. Until recently a graphics API was needed to code on GPUs which made coding for non graphics oriented calculations tough. Trying to work around this

limitation nVidia released CUDA which allows GPUs to be programmed using a variation of C. This enables a low learning curve and makes programming easier.

The three abstractions of the CUDA model are a hierarchy of thread groups, shared-memories, and barrier synchronization. Threads are arranged in the form of a grid which is a two dimensional array of thread blocks. Each thread block is a three dimensional structure that houses the threads. This type of hierarchy is given to the programmer so that the arrangement of the threads is similar to the way programmer's data is arranged (in arrays). Threads within a block can cooperate among themselves by sharing memory. Shared memory is expected to behave like an L1 cache where it resides very close to the processor core. Synchronization points can be specified by calling the function `__syncthreads`.

The memory available to the threads is of three types. Every thread has local memory. Number of threads which are in the same thread block can share memory. And the third type of memory is the global memory that every thread has access to. C code for both the GPU and the CPU resides in the same file. The CPU code follows a sequential flow. GPU code is called by a kernel call. This is where the code runs in parallel. A large number of threads are created by the kernel call. These threads then run parallelly on the GPU.

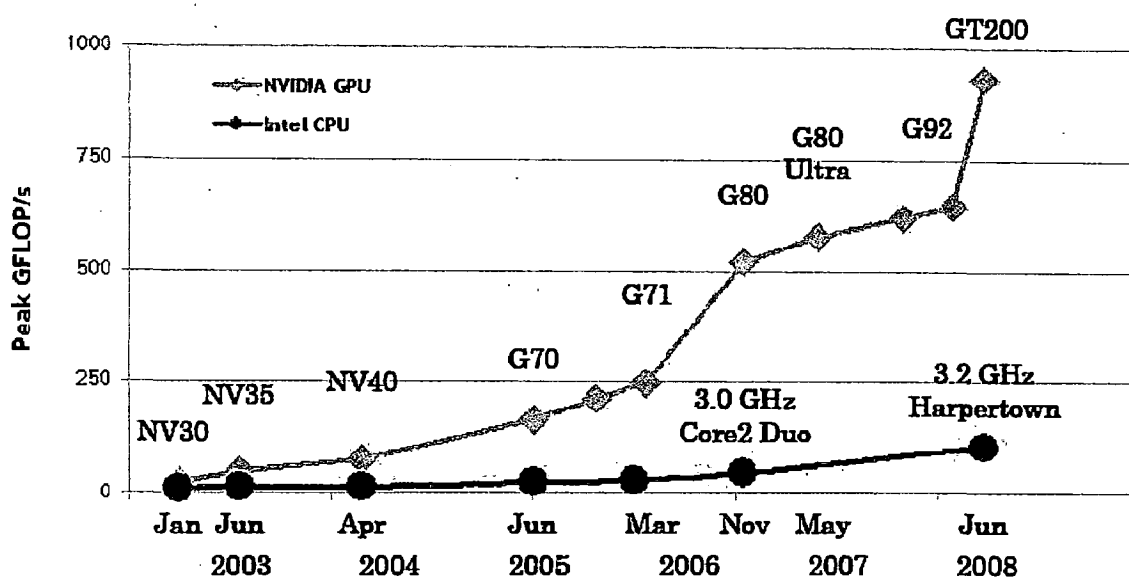


Figure 4.1: Floating point operations for the CPU and the GPU [16]

The figure 4.1 shows the tremendous computational capability of the GPU. GTX 280 a GT 200 family GPU delivers a peak performance of 933 GFLOPS/sec.

4.2.1 Programming Model

In this part we will discuss aspects that will explain how the CUDA programming model works and what the various aspects of the model are

4.2.1.1 Thread Hierarchy

Threads in CUDA are arranged in the form of a hierarchy. A number of threads house within what is known as a thread block. These thread block can be 1 dimensional, 2 dimensional or 3 dimensional. These thread blocks are placed in a structure known as thread grid. Thread grid can be either 1 dimensional or 2 dimensional.

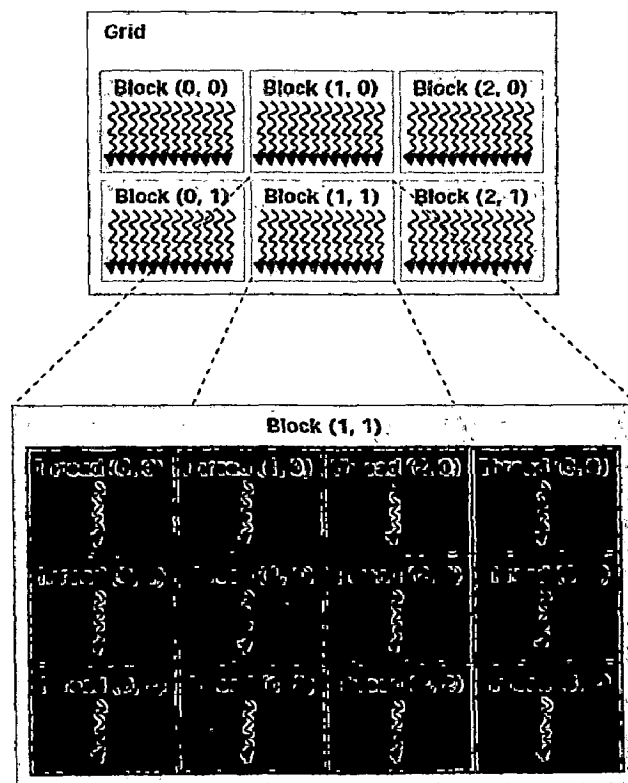


Figure 4.2: Figure showing arrangement of threads [16]

A maximum of 512 threads can be placed in a thread block. Thread blocks are expected to run independently of each other. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling scalable code to be written. Proper selection of grid size and block size is important to gain good speed up.

4.2.1.2 Memory Hierarchy

Threads may access memory from different memory spaces during their existence. Threads may declare local variable, may share memory with other threads that belong to the same block or may be accessing global memory.

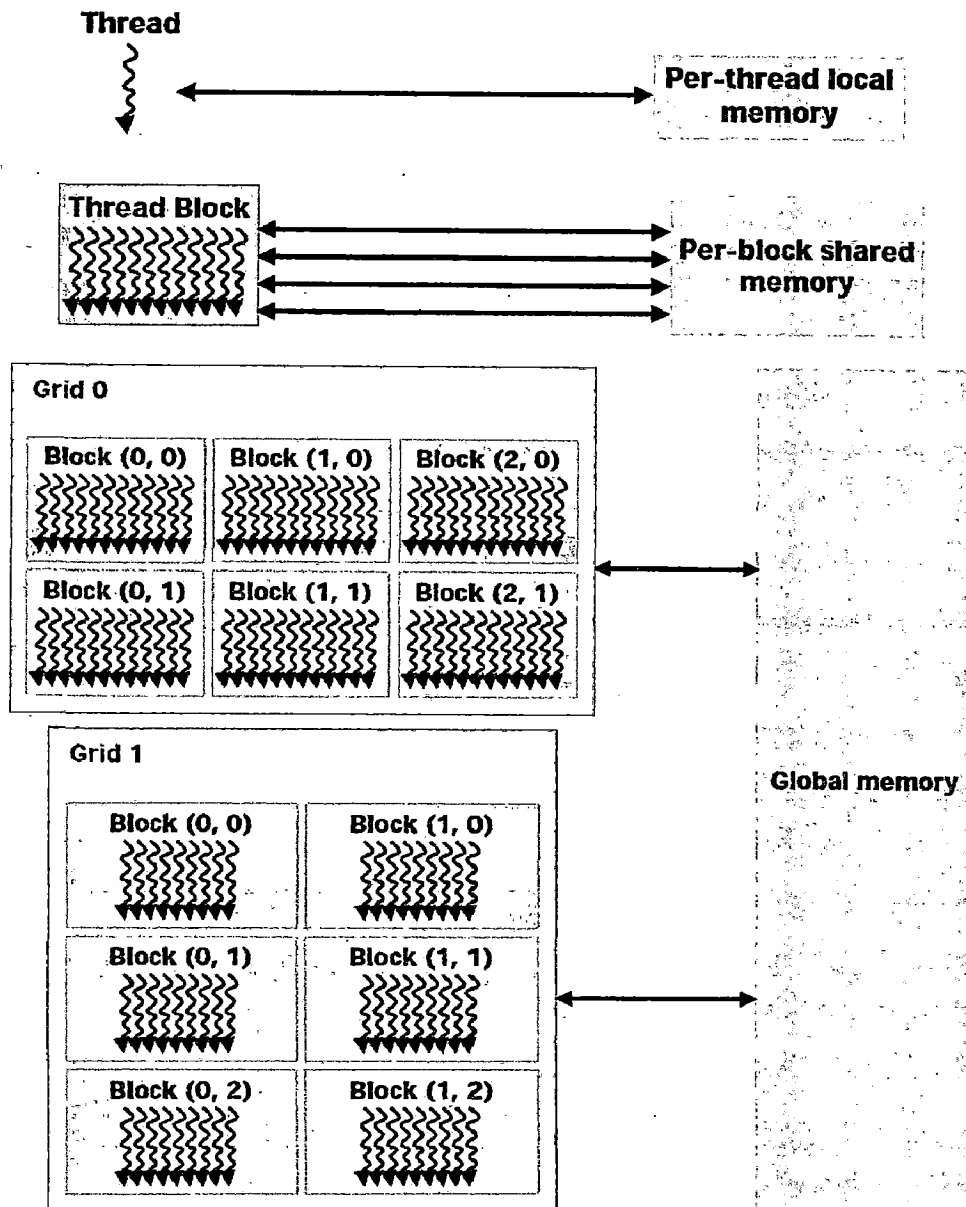


Figure 4.3: Figure showing how threads access global, shared and local memory [16]

4.2.2 GPU Implementation

In November 2006 nVidia significantly extended the GPU beyond graphics. It made available the massively parallel multithreaded GPU for general purpose applications. By scaling the number of processors and memory nVidia made available a wide range of products from the high ended GTX 280 with 240 cores and 1 GB RAM to 8400M GS with 16 cores and 128 Megabytes of RAM. The computing features enable a straightforward parallelization of the application by using C language. Some extensions have been made to C for CUDA specific code.

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors. A multiprocessor consists of eight Scalar Processor (SP) cores. Every multiprocessor has 8192 registers of 32 bit size each. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. The general idea is to achieve very fine grained parallelism by assigning one thread to work on one data item. A data element could be a pixel of an image or a protein base when working with poly peptide chains.

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address but are otherwise free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it splits them into warps. The way a block is split into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths.

A multiprocessor can work on a maximum of 8 thread blocks. However, if the thread code required a large number of registers then lesser number of thread blocks are assigned to a multiprocessor. In case a thread block is too bulky to be assigned to a multiprocessor then in such cases the kernel simply fails to launch.

4.3 EZW Algorithm

In this section we explain the general EZW algorithm. We use the same matrix used by Shapiro in his original paper [17]. The algorithm starts off by selecting a threshold (T). This threshold is the largest power of two that is smaller or equal to the maximum of all wavelet coefficients. A wavelet coefficient x is significant if $|x| > T$ and is insignificant otherwise. If a wavelet coefficient is insignificant at a coarser level then all children of that coefficient at finer levels will be insignificant.

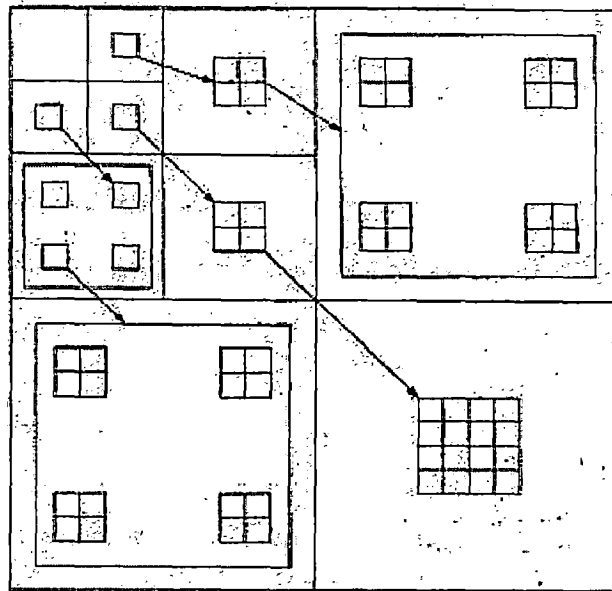


Figure 4.4: Parent Child Dependencies of Subbands of wavelet coefficients

The parent-child dependencies of subbands are shown in the Figure 4.4 above. The arrows point from the subband of the parents to the subband of the children. The top left subband represents the lowest frequency, the coarsest scale. The bottom right subband represents the highest frequency, the finest scale. Any coefficient is an element of a zerotree if all its descendants are insignificant with respect to the given

threshold value. If an element of a zerotree is not predictably insignificant from the discovery of a zerotree root at a coarser scale at the same threshold, it is called the zerotree root. The scanning of the wavelet coefficients is done either in Morton scan order or raster scan order. For our implementation we have used Morton scan order to read the wavelet coefficients.

Scanning of coefficients has two principles. The rule is that the coefficients at parent or coarser level are always scanned before proceeding to the children or finer levels. And the second rule is that coefficients of the same sub band are scanned before moving to the next subband. Figure 4.5 clearly shows how the scanning proceeds in Morton scan order.

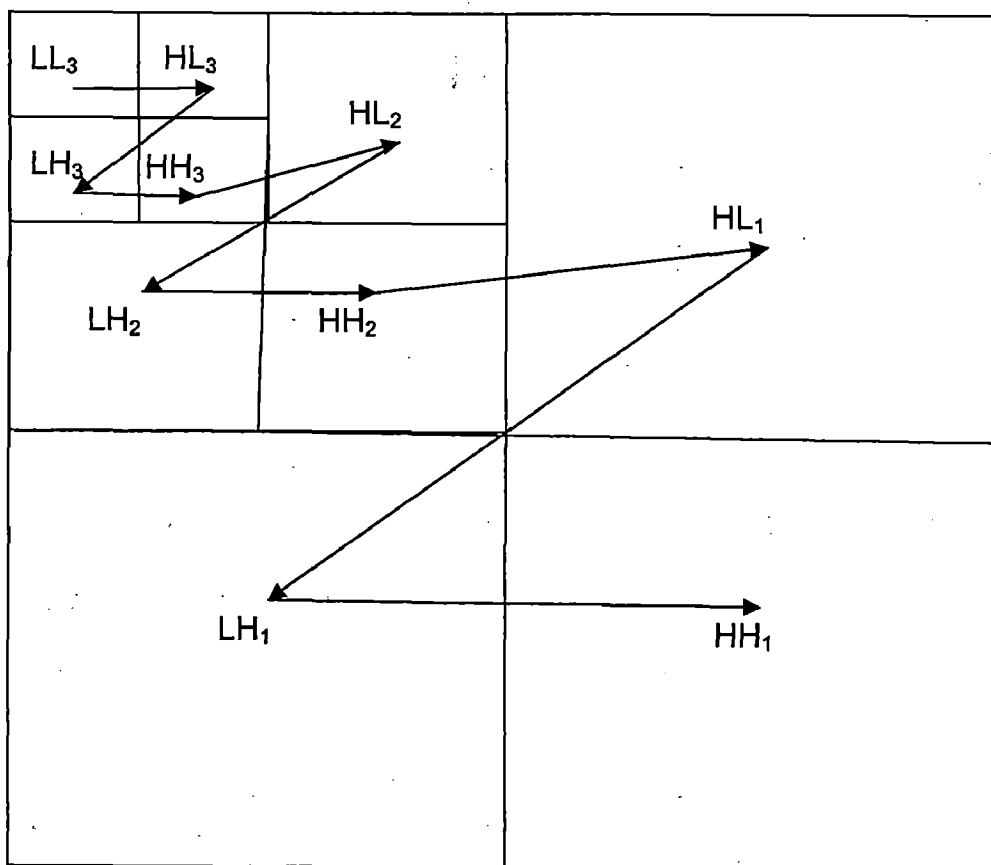


Figure 4.5: Morton Scan order on a matrix of wavelet coefficients at three level decomposition

The EZW algorithm has basically two parts the dominant pass and the subordinate pass

```

Threshold=2^floor(log2(max(all wavelet coefficients)))
While(Threshold>0)
{
    Perform dominant pass(Threshold)
    Perform subordinate pass(Threshold)
    Threshold=Threshold/2
}

```

Figure 4.6: EZW Algorithm

Dominant pass performs the task of encoding each wavelet coefficient as any of the following 4 symbols

1. Zero tree root (t): This symbol is encoded if the coefficient is insignificant when compared with the threshold and also if all the descendants of the coefficient are insignificant when compared with the threshold. If such a symbol is encountered then none of the children of this coefficient are considered for this dominant pass
2. Isolated zero (z): This symbol is encoded if the coefficient is insignificant when compared with the threshold but some descendant of the coefficient is significant when compared with the threshold
3. Positive significant (p): This symbol is encoded if the coefficient is significant when compared with the threshold and is positive in sign.
4. Negative significant (n): This symbol is encoded if the coefficient is significant when compared with the threshold and is negative in sign.

We now take a specific case to explain how dominant pass works. The largest of all the wavelet coefficients is 63. This makes the threshold for the first dominant pass as 32. The scan starts at 63 which is encoded as a positive significant as it is greater than 32. This is followed by -34 which is negative in sign and is therefore encoded as negative significant. Next we read -31. This value is less than the threshold but one of its children 47 is greater than the threshold so -31 is encoded as isolated

| | | | | | | | |
|-----|-----|-----|-----|---|----|-----|----|
| 63 | -34 | 49 | 10 | 7 | 13 | -12 | 7 |
| -31 | 23 | 14 | -13 | 3 | 4 | 6 | -1 |
| 15 | 14 | 3 | -12 | 5 | -7 | 3 | 9 |
| -9 | -7 | -14 | 8 | 4 | -2 | 3 | 2 |
| -5 | 9 | -1 | 47 | 4 | 6 | -2 | 2 |
| 3 | 0 | -3 | 2 | 3 | -2 | 0 | 4 |
| 2 | -3 | 6 | -4 | 3 | 6 | 3 | 6 |
| 5 | 11 | 5 | 6 | 0 | 3 | -4 | 4 |

Figure 4.7: Wavelet coefficients of an 8 x 8 image [17]

zero. 23 is encoded as a zero tree root since all its children are less than the threshold. After this we move to the HL2 coefficients. In these 49 is encoded as a positive significant while all others 10, 14 and -13 are zero tree roots. In the LH2 coefficients all except 14 are zero tree roots. 14 is an isolated zero as its child 47 is greater than the threshold. We do not encode any HH2 or HH3 coefficient as their parent 23 is a zero tree root. Among all the LH1 coefficients -1, 47, -3, 2 are encoded since only their parent was encoded. 47 is a positive significant while all others are zero tree roots. So our output after the first dominant pass is pnztptttztttttptt.

Dominant pass is followed by subordinate pass. In this a binary string is stored in the file. The subordinate pass only looks at the nonzero values and refines them. It basically tries to store information about whether the coefficient lies in the upper half or the lower half. For this pass when the threshold is 32 all coefficients that have been encoded with a 'p' or an 'n' will have a value greater than 32 but less than 64. They have to be less than 64, else the threshold would have been 64 itself. The range [32, 64) is partitioned into [32, 48) and [48, 64). All values lying in [32, 48) are encoded as a '0' and those lying in [48, 64) are encoded as a '1' in the subordinate pass. The output after the subordinate pass is 1010. After this the threshold is halved and then similar iterations of dominant and subordinate passes follow.

4.4 Our Algorithm

Straightforward parallel implementation of EZW is not possible because the coefficients are needed to read in a sequence and each of these coefficients cannot be worked on in parallel. We have solved the EZW algorithm by creating three trees and solving them in parallel. The creation of the three trees is also done in parallel. Care has to be done so as to ensure that same code works on all three trees. We will use the same example of 8 x 8 image to explain its working on our example.

Our tree structure for every tree is such that it is a complete tree with every node having four children. We will refer to such a tree as a quaternary tree for our discussion. The three trees created are in three directions of details: horizontal, vertical and diagonal. For now we leave encoding of the dc coefficient 63 as its encoding depends on the codes of its children: -34, -31 and 23.

The first step is to create the trees. The trees are separate in just their orientation. The coefficients are to be read in Morton order only. The trees we get are

| | | | | | | | | | | | | | | | | | |
|-----|-----|----|----|-----|-----|---|---|----|---|----|---|----|---|---|---|---|--|
| L 0 | -34 | | | | | | | | | | | | | | | | |
| L 1 | 49 | 10 | 14 | -13 | | | | | | | | | | | | | |
| L 2 | 7 | 13 | 3 | 4 | -12 | 7 | 6 | -1 | 5 | -7 | 4 | -2 | 3 | 9 | 3 | 2 | |

| | | | | | | | | | | | | | | | | | |
|-----|-----|----|----|----|----|----|----|---|---|----|---|----|---|----|---|---|--|
| L 0 | -31 | | | | | | | | | | | | | | | | |
| L 1 | 15 | 14 | -9 | -7 | | | | | | | | | | | | | |
| L 2 | -5 | 9 | 3 | 0 | -1 | 47 | -3 | 2 | 2 | -3 | 5 | 11 | 6 | -4 | 5 | 6 | |

| | | | | | | | | | | | | | | | | | |
|-----|----|-----|-----|----|----|---|---|---|---|---|---|---|---|---|----|---|--|
| L 0 | 23 | | | | | | | | | | | | | | | | |
| L 1 | 3 | -12 | -14 | 8 | | | | | | | | | | | | | |
| L 2 | 4 | 6 | 3 | -2 | -2 | 2 | 0 | 4 | 3 | 6 | 0 | 3 | 3 | 6 | -4 | 4 | |

Figure 4.8: Figure showing the trees built for the horizontal, vertical and diagonal details.

Here L0, L1 and L2 stand for levels 0, 1 and 2. This type of tree arrangement as a two dimensional array helps in a simple implementation for CUDA. The size of every tree is fixed from the beginning as every node will have 4 children and the depth of the tree is a direct function of the size of input image. Also locating of parent node and children node is simple. The size of one tree can hence be computed. Also locating parent node and children node is simple. For any node p at level i

Parent (p) = (p/4) at level (i-1) for $i \neq 0$

Children (p) = (4p), (4p+1), (4p+2), (4p+3) at level i+1

These three trees are solved for their individual dominant passes level wise. After solving for threshold 32 in parallel and we get the following strings

Tree 1: nptttttt

Tree 2: ztzttptt

Tree 3: t

Once these are solved in parallel we merge these three trees by reading elements from every tree. We read exactly 4^k elements from each of the trees where k starts from 0. So we read one element each individually

Resultant String: nzt

After this we increase.k by 1 so this means that now we have to add 4 elements from each of the tree. At this point tree 3 is complete

Resultant String: nztptttzt

Again k is increased by 1 and now we need to read 16 elements from each tree. However only 4 are left which will be added to the string

Resultant String: nztptttztzzttttptt

This string is same as that obtained when a dominant pass was ran over the entire matrix. To hide some of the latency in this process we perform the dominant pass on the GPU and the merging of these results on the CPU in parallel. So when dominant passes for threshold 16 are run on the GPU results of dominant pass for threshold 32 are merged. This actually becomes more logical as there is always one dominant pass more than the number of subordinate passes. The last subordinate pass is of no importance because the threshold is 1 and every output of subordinate pass will be a 0.

The parallelizations achieved during in this approach are

1) Dominant Passes

The dominant passes are performed level wise on all the three trees. The logic of performing the dominant passes as three trees has been explained in section 4. To make the process further parallel we solve the three trees level wise.

```
For(i=0;i<levels;i++)  
    Dominantpass(trees,3*pow(4.0,i));
```

By doing so we gain more speed up. We first create just three threads to get symbols for all coefficients at L0 for the three trees. This is followed by creation of 12 threads to get symbols for all coefficients at L1 for all three trees. Likewise we get a general progression to generate $3 * 4^k$ threads to get symbols for all coefficients at level k. This basically increases the amount of parallelism as the size of the array becomes larger. For image size of 128x128 we have as many as 12288 threads at the last level.

2) Subordinate Pass

Another area of computation gain is the performing of the subordinate pass itself. The entire vector is processed in parallel This vector gets up to around 16000 elements for a 128 x 128 image which results in as many number of threads. The subordinate pass is a set of simple steps that put a 1 or a 0 for every significant wavelet coefficient.

```
Subordinate (list_d, threshold, output)
```

```
For every thread i
```

```
if((abs(list_d[i])&(threshold))!=0)
    output[i]='1';
else
    output[i]='0';
```

3) Writing to file

Writing into files is a time consuming process. Ideally we should have lesser number of file writes to increase speed up. We do this by using just one file write call instead of a number of character by character writes. For doing this we process the bit string in parallel and make it into a long string of characters and we write this string in one go instead of a number of character by character writes. This does change the implementation a little bit. The streaming character of the algorithm is lost as we now wait for the entire string to be ready to be written.

4.5 Results

The results were obtained on a system running on a Intel Core 2 Quad 2.4 GHz processor with 2.75 GB RAM. The graphics card on the system was GeForce GTX 280. This particular GPU has 240 stream processors. The coding was done in Visual Studio 2005. The experimental results show the amount of time required for EZW encoding of images starting from sizes 128 x 128 to 1024 x 1024. We have not counted the times that the algorithm takes for console output as that itself becomes more expensive than the EZW encoding.

The parallel implementation of EZW turns out to be slower for smaller images as the vectors are not long enough for effective parallelization. The stitching of the trees takes up significant amount of time and is something that cannot be done in parallel. However, as the image sizes grow parallelization starts becoming effective and gradually tries to reach the ideal speed up of 3, which is the number of trees that have been solved in parallel.

| Image Size | Our Algorithm (on CUDA) | Standard Algorithm |
|-------------|-------------------------|--------------------|
| 128 x 128 | 32 ms | 31 ms |
| 256 x 256 | 110 ms | 141 ms |
| 512 x 512 | 406 ms | 657 ms |
| 1024 x 1024 | 1703 ms | 2953 ms |

Table 4.1: Table showing the relative performance of our parallel algorithm against a standard linear implementation

| Image Size | Speed Up |
|-------------|----------|
| 128 x 128 | 0.97 |
| 256 x 256 | 1.28 |
| 512 x 512 | 1.61 |
| 1024 x 1024 | 1.74 |

Table 4.2: Table showing the corresponding speedups at different image sizes

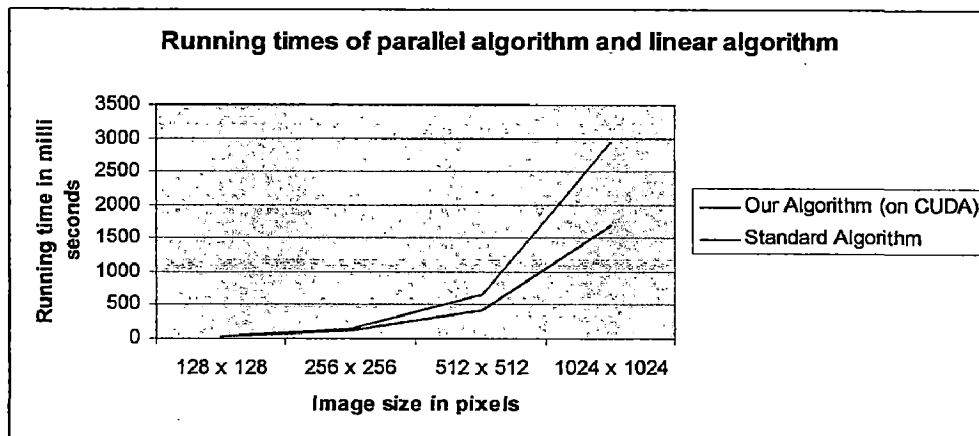


Figure 4.9: Comparison of running times of parallel implementation against linear implementation

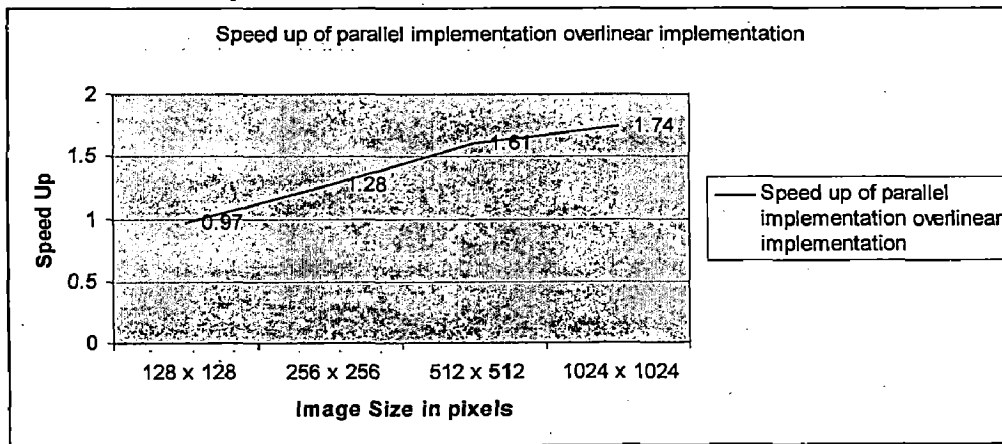


Figure 4.10: Speed up of parallel implementation over linear implementation with varying image sizes.

4.6 Conclusions

We have presented a fresh approach to solve the latency issue of EZW encoding by utilizing the computation power of the GPU. We do the encoding in parallel based on the observation that the three trees for horizontal, vertical and diagonal details of wavelet coefficients are independent. For an image of 1024 x 1024 we were able to get a speed up of 1.74 times over the linear implementation. For an additional GPU we are able to gain a speed up of 1.74.

Chapter 5 Background Modeling on MultiCore Processors

The aim of this chapter is to compare two implementations of single Gaussian on two multiprocessors. The implementations have been done on Cell BE processor (CBE) and on the GPU using nVidia's CUDA architecture. We will give the algorithm for performing background modeling using the single Gaussian and then we will go into the details of how this is parallelized on the two processors.

5.1 Background modeling using Single Gaussian

Background modeling uses one Gaussian per pixel. Information about the first frame is used to initialize the Gaussians' means. The standard deviations are set to zero.

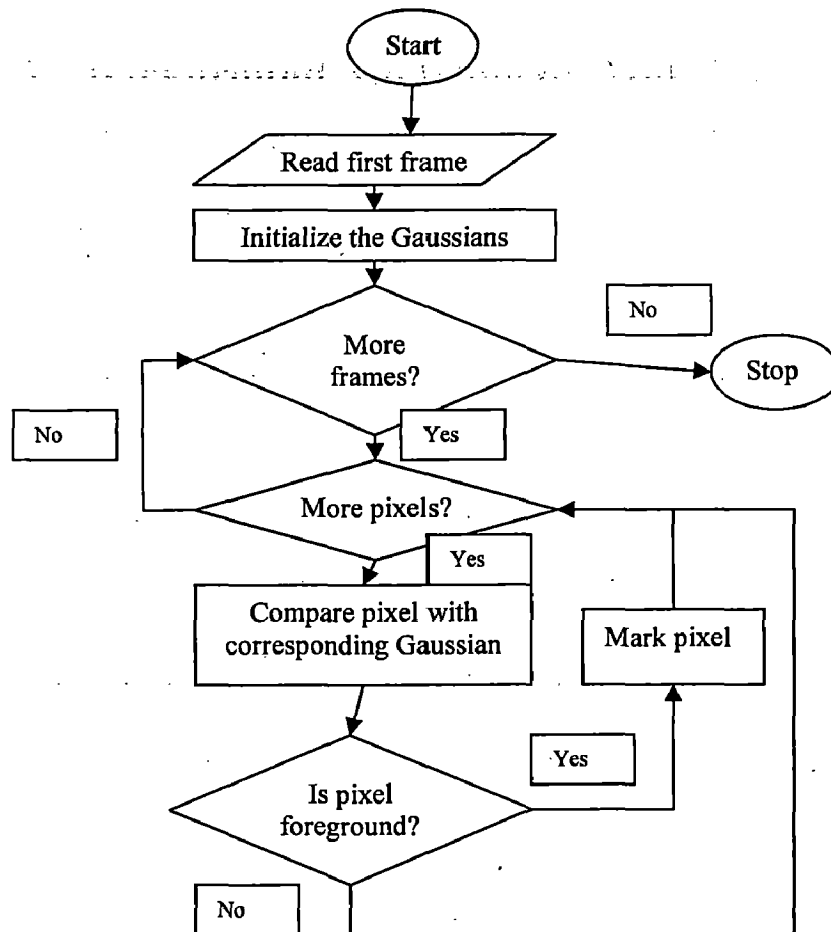


Figure 5.1: Flowchart for background modeling

After this every pixel of the incoming frame is compared to the corresponding Gaussian. If the value of the pixel lies within 2.5 standard deviations of the mean then the pixel is accepted as a background pixel. The idea is that we store information for every pixel which can be considered as a mathematical model of the background pixel process. Now if some pixel shows a lot of deviation from this model then the pixel can be safely considered as a foreground pixel. The choice of selecting the spread of 2.5 standard deviations about the mean follows from the theory of normal distribution. The pixel values are assumed to follow normal distribution. The theory tells that 95 % of values fall within 2 standard deviations of mean and 3 standard deviations account for 99.7% of values [21]. So, 2.5 standard deviations should lie somewhere in between 95% and 99.7%. The idea is that the camera has some acquisition noise and by 2.5 standard deviations difference we are allowing some leniency in the system to compensate for this acquisition noise.

If the difference about the mean is very less then we will have a situation where there will be many false detections of foreground pixels as acquisition noise of the camera itself shall be enough to be detected as a foreground pixel. If the difference about the mean is too great then a situation can arise where actual foreground pixels too are not getting counted as foreground pixels.

The Gaussian per pixel is needed to be updated to take into account the changes that are gradually coming about in the setting. This has to be done to allow an object to become part of the background if it stays stationary for some time. Also if some object that was stationary and then start to move, then this object should be later counted as a foreground object. By updating of the Gaussian we mean updation of the mean and standard deviation of the Gaussian.

The background model for each pixel is then updated according to the pixel value in the newer frame. If μ is the mean of the background model and σ is the standard deviation and x be the pixel under consideration of the incoming frame.

$$\mu \leftarrow (1 - \alpha)\mu + \alpha x$$

$$\sigma^2 \leftarrow \max(\sigma_{\min}^2, (1 - \alpha)\sigma^2 + \alpha(x - \mu)^2)$$

Here α is the learning rate. More the value of α quicker the system updates the background model with the incoming frames. σ_{\min} is a value that is forced by the system so to ensure that the value of standard deviation does not become too small to accept background pixels.

5.2 STI Cell Broadband Engine

We now discuss a little bit about the STI Cell Broadband engine that we will be using to parallelize background modeling. STI stands for Sony, Toshiba and IBM Corporation. Cell BE is an outcome of alliance between the mentioned companies in the year 2001. In year 2004 first operational Cell BE engine was released. Cell BE engines also found their way in the popular gaming console PlayStation 3. The CBE processor is the first implementation of a new family of multiprocessors conforming to the Cell Broadband Engine Architecture which extends 64 bit Power PC Architecture. The CBE processor was intended for heavy graphics usage applications like gaming console but the architecture has enabled a broad range of applications to gain performance.

The CBE processor is a heterogeneous process with two type of processing elements. Their function is specialized into two types: the Power Processor Element (PPE) and the Synergistic Processor Element (SPE). The CBE processor has one PPE and eight SPE's.

5.2.1 Hardware Architecture

The hardware comprises of the following components

5.2.1.1 PowerPC Processor Element (PPE)

The PPE contains a 64-bit PowerPC Processor Unit (PPU) with associated caches that conform to PowerPC Architecture reduced instruction set computer (RISC) core with

a traditional virtual memory subsystem. It runs an operating system, manages system resources, and is intended primarily for control processing, including the allocation and management of SPE threads. The PPE consists of two main units Power Processor Unit (PPU) and PowerPC Processor Storage Subsystem (PPSS). The PPU performs instruction execution, and it has level 1 (L1) instruction cache, data cache of 32KB each, and six execution units. The PPSS handles memory requests from PPU and external requests to the PPE from SPEs or I/O devices. It has a unified level 2 (L2) instruction and data cache of 512KB.

The primary function of the PPEs is the management and allocation of tasks for the SPEs in a system. When data enters the PPE, this element then distributes it among SPEs, schedules them to be processed on one or more of the SPEs, controls and synchronizes them.

5.2.1.2 Synergistic Processor Elements (SPEs)

Each of the eight Synergistic Processor Elements (SPEs) is a 128-bit RISC processor specialized for data-rich, compute-intensive SIMD applications. It consists of two main units, the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). The SPU deals with instruction control and execution. It includes a single register file with 128 registers (each one 128 bits wide) and a unified (instructions and data) 256-KB local store (LS). Each SPU is an independent processor with its own program counter and is optimized to run SPE threads spawned by the PPE. The SPU fetches instructions from its own LS, and it loads and stores data from and to its own LS.

5.2.1.3 Element Interconnect Bus (EIB)

The EIB is a communication bus internal to the Cell processor which connects the various on-chip system elements: the PPE processor, the memory controller (MIC), the eight SPE coprocessors, and two off-chip I/O interfaces, for a total of 12 participants. The EIB also includes an arbitration unit which functions as a set of traffic lights.

5.2.2 Software Development Kit

An SDK is available for the Cell Broadband Engine. The SDK contains the essential tools required for developing programs for the Cell Broadband Engine. The SDK consists of numerous components including the following

- The IBM Full System Simulator for the Cell Broadband Engine, systemsim.
- System root image containing Linux execution environment for use within systemsim.
- GNU tools including c and c++ compilers, linkers, assemblers and binary utilities for both PPU and SPU.

5.3 Parallelization on CBE

The basic idea when parallelizing background modeling is to split the image into eight parts and work on each part on one SPE. On CBE the code for PPE and SPE is written in two separate files. We will explain the work done by both PPE and SPE in two following sections.

5.3.1 Work done on PPE

The main work of the PPE is to read the images for the SPEs and to coordinate the SPEs. The images that were taken had a resolution of 100 x 100 for a total of 10000 pixels. So the task to be assigned to every SPE was to work on 1250 pixels. For this the required pixels were first copied to the local store of every SPE.

The first task done by the SPE is to copy 8 parts of the first frame to corresponding SPE. This is used to create a background model. These values are used by the SPEs for future processing. CBE enforces that every memory access be byte aligned. For our task we used 128 byte alignment. So instead of the original 1250 pixels of data every SPE was sent 1280 bytes of data. We padded up the last 30 pixels worth data with zeros to keep memory aligned.

After this 8 SPE threads are created, following which SPEs spring into action. The working of SPE will follow this section. After creating these 8 threads, PPE proceeds to read the next file. This data is also padded to keep the memory addresses aligned. When the PPE has finished read the file, it signals to the SPEs about the availability of next data and waits onto the mailbox. The PPE is waiting for the SPEs to send back the background modelled information so that it can write this into an output file. When the SPEs have finished their background modeling they signal the PPE to proceed with writing the image.

The PPE after getting the data from the SPEs writes the data sequentially into an output file and then proceeds to read the next image.

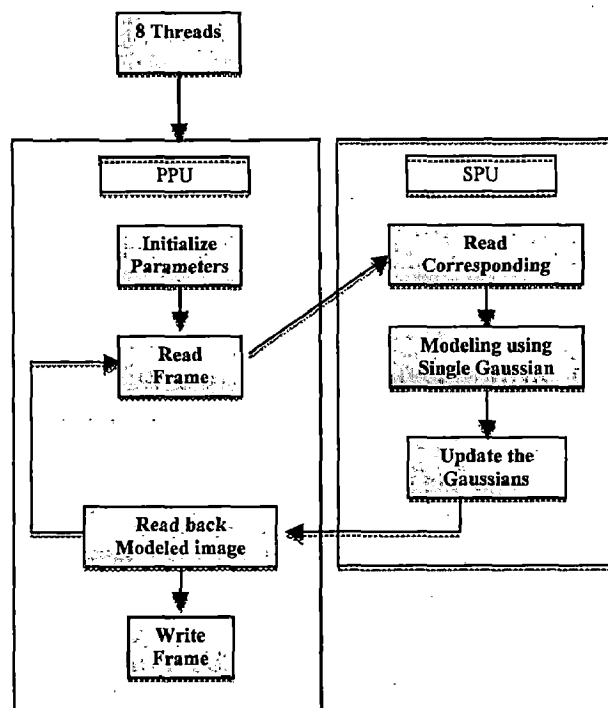


Figure 5.2 Parallelization of background modeling on CBE

5.3.2 Work done on SPE

The responsibility of the SPE is to perform the actual background modeling as well as to read the information from the PPE and write the background modelled image part back to the PPE.

First thing that SPE code performs is to identify which SPE it is. This is done by reading an environment variable. This number is used to identify as to from which array this SPE has to write the data to. Data is read using `mfc_get` from the arrays. Since data parallelization is done, by simply using the SPE number the SPE reads the corresponding part of the image. Since the data has been put as 128 byte aligned, each SPE reads 1280 pixels worth of data instead of the planned 1250 pixels. However the processing is done for just 1250 pixels of the data.

The SPU first reads the first frame and stores the information from this frame to make its background frame. After this the background modeling process starts where the SPE code sits in a loop that loops over all images. The loop starts with the SPE waiting on mailbox. The SPE waits for the PPE to read the image and signal it. Once the SPE knows that the data is available then the SPE reads the corresponding part of image on which it is supposed to work on. After getting the data the actual background modeling starts. The image pixels are compared against the background model to see whether they lie within 2.5 standard deviations of the mean of the model. If the pixel falls outside 2.5 standard deviation then the pixels has to be marked otherwise the pixel is left as it is.

The means and standard deviations are updated to update the model with the updated environment. The implementation uses a learning rate of 0.95. Updation of means is straightforward however updation of standard deviation is slightly different. A minimum value is forced for standard deviation. This is done so that the deviation does not become a very small number. If it so happens that the deviation becomes very small, then the acquisition noise of camera shall be enough to cause false positives. The data is written back using the `mfc_put` call. After writing the SPE informs the PPE that it can accept the next frame for processing and following this the SPE starts to wait for data to come from the PPE.

5.4 Parallelization on GPU

The focus when parallelizing using nVidia CUDA architecture is to process every pixel in parallel. The code for both the CPU and the GPU is in the same file unlike the case with CBE. nVidia GTX 280 has 240 cores on its 30 multiprocessors. The idea is

to create one thread for every pixel. The algorithm proceeds to see if the pixel is background or foreground.

The first frame is read on the CPU. Memory for storing the background model is allocated on the GPU using `cudaMalloc` function call. `cudaMalloc` allocated one dimensional memory on the GPU. The first frame is then copied to the GPU memory by using the function call `cudaMemcpy`. In the case of CUDA `cudaMemcpy` is synchronous, that means that the control does not return to CPU till the copy operation is complete. `cudaMemset` is used in cases to initialise a GPU memory block with a particular value. This is used to set the initial standard deviations of the background model to zero.

The CPU code then proceeds to loop over all the images. The CPU reads every image one by one. After this the CPU proceeds to copy the image to the GPU using `cudaMemcpy`. The images that we have taken for background modeling are 100 x 100 pixels. The kernel call creates 10000 threads and passes appropriate parameters.

The first line of the CUDA thread identifies the thread number that is running. The parallel runs of threads shall not create any race problems as there is one thread per pixel. Each thread shall update only its memory so no race conditions will come up. This thread number is to access and update appropriate memory. The means and standard deviations are updated in a fashion similar to that on CBE. When the kernel finishes control proceeds on the CPU. Actually, kernel calls are asynchronous and pass the control back to CPU after initiating the call. Since there is no useful work that can be done on the CPU during this time, the CPU is blocked from proceeding by using `cudaThreadSynchronize` call. This call blocks the CPU till all previous CUDA calls are complete. When the control proceeds it is assured that the GPU has finished background modeling. Then the CPU initiates a `cudaMemcpy` to copy the background modelled image back to the CPU. After this the CPU writes the image character by character. The loop over all the images continues this process.

5.5 Results and Conclusions

We now give the results when background modeling was performed using a single Gaussian. The implementations were coded in C and ran in Visual Studio 2005. The running times were calculated only for the part where background modeling was performed and file I/O time was not considered as the time was these operations itself is higher than the code for background modeling. The results are presented when background modeling was performed over 150 images of size 100 x 100 pixels.

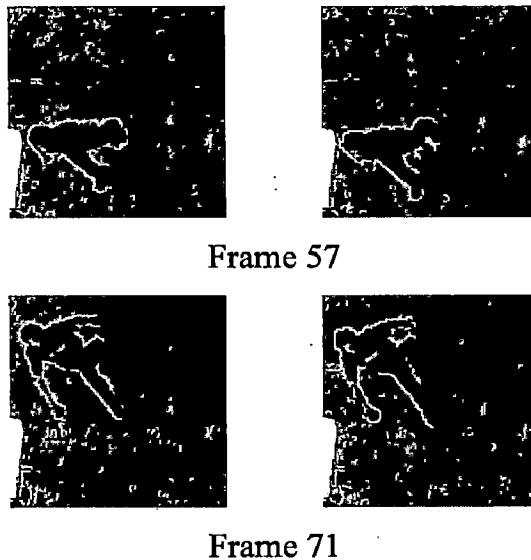


Figure 5.3: Input and the corresponding background modelled images

| Processor | Time in milliseconds |
|---------------------------|----------------------|
| Linear (Core2Duo 1.5 GHz) | 1329 |
| CBE | 210 |
| CUDA (GTX 280) | 79.5 |

Table 5.1: Table showing the running times of background modeling for three implementations

The speedup for CBE was 6.3 while the same for CUDA was 16.7. The speed up for CUDA was better because of the thread level parallelism that offered 10000 parallel threads to be run on the 240 cores of GTX 280. The implementation for CBE was not the most optimum implementation as it lacked feature like double buffering which could have enhanced CBE's performance.

In this chapter we will cover parallelization of 3 algorithms on nVidia's CUDA architecture. We will talk about parallelization of Gaussian Mixture Model (GMM), Morphological operation and Connected Components Labelling (CCL). These three algorithms are used for performing object tracking.

6.1 Introduction

We will briefly talk about what these operations do and what is their role in the entire video surveillance module.

GMM is a background modeling algorithm. It uses more than one Gaussian per pixel to handle dynamic backgrounds. By using more than one Gaussian more information is retained about every pixel and if there is some background object that is dynamic like moving trees or flag then those pixels can be caught as background pixels. Morphological operations are used to fill up noisy pixels in an image. After performing GMM the image has some false positives. Further some pixels that actually are foreground pixels may not have gotten marked as foreground. So to make this distinction clear morphological operations like dilation and erosion are coded. The idea is to make the video surveillance module more robust. After applying appropriate morphological operations CCL is run over the image. CCL labels and groups pixels into logical objects. So after CCL we get the number of objects that are there in the image. We can put a bounding box across these objects and this can give visual information about where the object is. It can be extended to find the path the object is taking.

The figure 6.1 shows the changes the image undergoes when the three operations are performed. The first image has some irregularities that could be because the GMM algorithm was not able to filter out the pixels correctly. Such a thing is expected in a field like computer vision. However, application of morphological operations fixes up these outliers. CCL then proceeds to link these pixels into logical object.

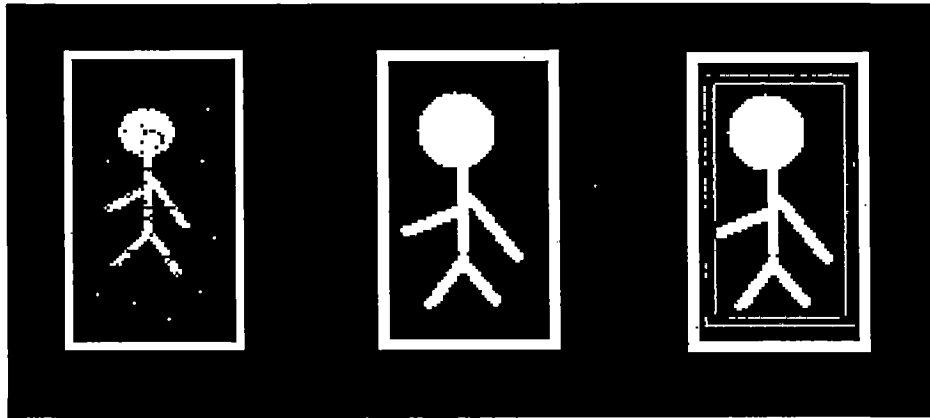


Figure 6.1: Application of GMM, morphological operations and CCL algorithms

6.2 Background modeling and detection of foreground and background regions

This step forms the bulk of computation which varies depending on the complexity and robustness of the algorithm used. We use pixel-level Mixture of Gaussians (MOG) background model which has been used in a wide variety of systems because of its efficiency in modeling multi-modal distribution of backgrounds (such as waving trees, ocean waves, light reflection, etc) and its ability to adapt to a change of the background (such as gradual light change, etc.). It models the intensity of every pixel by a mixture of K Gaussian distribution and hence becomes computationally very expensive for large image size and value of K . Furthermore, there is high degree of data parallelism in the algorithm as it involves independent operations for every pixel. Thus, compute intensive characteristic and available parallelism makes MOG suitable candidate for parallelizing on multi-core processors.

Gaussian mixture model is a more robust algorithm for background modeling when compared to single Gaussian. In addition to holding mean and standard deviations the Gaussians now hold a weight as well to distinguish the Gaussians among themselves. This is also used to figure out the relative importance of the Gaussians. Parallelization of GMM too works like single Gaussian where one thread is allocated to work on every pixel.

GMM threads are much heavier than single Gaussian threads as the amount of task done is much more. Now updations of Gaussians involve updations of K Gaussians.

Also all check are performed against K Gaussians. Further GMM has another concept where the number of Gaussians to be involved in background modeling can be changed. We suggest the reader to refer to [4] for details.

For parallelization the choice of data partitioning is to either break the data in form of columns, rows or tiles. GPU have a high number of cores that allow for independent threads to be scheduled on separate cores. Since GMM offers pixel-level data parallelism, it is very efficient for parallelization on CUDA. Figure 6.2 shows the tile structure on CUDA where every pixel is processed by one thread. One thread is invoked per pixel. So for an image of 320×240 pixels we have 76800 threads which are invoked in parallel. Programming on CUDA has no local memory limitations (only limited by the actual DRAM on the system). This allows for the entire image to be copied in one go from the DRAM to GPU memory.

| | | | |
|--------|--------|--------|--------|
| i | $i+1$ | $i+2$ | $i+3$ |
| $i+4$ | $i+5$ | $i+6$ | $i+7$ |
| $i+8$ | $i+9$ | $i+10$ | $i+11$ |
| $i+12$ | $i+13$ | $i+14$ | $i+15$ |

Figure 6.2: Tile structure on GPU where every pixel is processed by one thread

The running times for GMM implementations on linear as well CUDA architecture are given below. The running time is for 1 frame of size 320×240 size. The speed up of the parallel implementation is 7.59 times.

| | |
|------------|-----------|
| CUDA | 0.586 ms |
| Sequential | 4.4511 ms |

Table 6.1: Table showing the running time of GMM on image size of 320×240 pixels for sequential and parallel implementation

6.3 Binary morphological operations

Morphological operations process an input image by applying a structuring element and producing an output image where each pixel is based on a comparison of the corresponding pixel in the input image with its neighbours depending upon the size and shape of the structuring element. The most basic morphological operations are dilation and erosion and all others like opening, closing etc. consist of some combination of these two operations. Dilation, denoted by the operator \oplus , adds pixels to the boundaries of objects in an image, while erosion, denoted by the operator \ominus , removes pixels on object boundaries. The rule for dilation is that the value of the output pixel is the maximum value of all the pixels in the input pixel's neighbourhood. In a binary image, if any of the pixels is set to the value 1, the output pixel is set to 1. The rule for erosion is that the value of the output pixel is the minimum value of all the pixels in the input pixel's neighbourhood. In a binary image, if any of the pixels is set to 0, the output pixel is set to 0. This is described mathematically as

$$A \oplus B = z \left\| \left(\hat{B} \right)_z \cap A \neq \phi \right.$$

$$A \ominus B = z \left\| (B)_z \subseteq A \right.$$

Where \hat{B} is the reflection of set B and $(B)_z$ is the translation of set B by point z as per the set theoretic definition.

Our implementation is restricted to binary morphological operations. Morphological elements have a structuring element. Structuring element can be understood as a window that is slid over the image to perform morphological operations.

The center of this element is placed over that element on which the operation is to be performed. The difference between dilation and erosion is very small. In case of dilation if any element of the structuring element and the corresponding element of

| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

Figure 6.3: A 3 x 3 structuring element

the image is 1 then the location where the center pixel of the structuring element is placed is changed to 1. So it is basically searches nearby pixels and if the region is foreground it enhances the region. This is used in cases where foreground objects have some anomalies. This enhances the objects and makes them complete. On the other hand, erosion is used to suppress false positive foreground detections. The erosion operation searches for 0s in both the structuring element and the image.

Morphological operation too offer a per pixel parallelization like the GMM algorithm. However there can be an issue with threads running in parallel. Since the structuring elements of neighbouring pixels may overlap, we may get a race condition with threads trying to update a pixel that may not have been read till now. The parallel implementation was done in a straightforward convolution way by running parallel thread for each pixel. However on the boundary pixels due to overlap, we may get a race condition as shown in figure 6.4 with threads trying to update a pixel that may not have been read till now. To solve this problem we allocate a different memory that is written on the GPU and then this memory is copied back to the CPU. In the kernel we run a loop that traverses the structuring element one by one checking the neighbouring region of the pixel the thread is working upon. If the kernel is placed on a region that does not belong to the image then that memory access is stopped. This happens for the boundary pixels of the image. If the kernel size is larger, then the time required for morphological operation increases as the iterative structure of the kernel causes a significant delay.

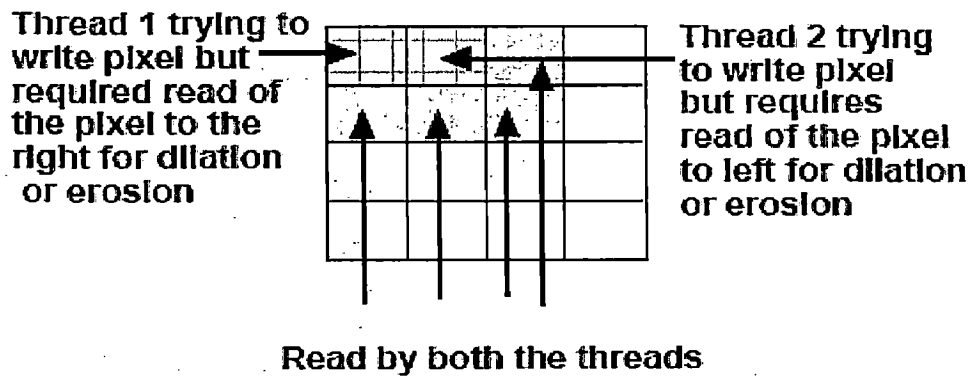


Figure 6.4: Figure shows the race conditions on boundary pixels

The table below shows the time required to perform on morphological operation on an image of size 320 x 240 pixels

| Structuring Element | | Linear Implementation | GPU Implementation Code | |
|---------------------|-----------|-----------------------|-------------------------|----------|
| Size | Shape | | Time | Speed Up |
| 7 x 7 | Ellipse | 89.806 ms | 7.8 ms | 11.5 |
| 5 x 5 | | 47.74 ms | 4.5 ms | 10.6 |
| 3 x 3 | | 16.324 ms | 3.2 ms | 5.1 |
| 7 x 7 | Rectangle | 24.235 ms | 8.3 ms | 2.91 |
| 5 x 5 | | 18.782 ms | 4.1 ms | 4.5 |
| 3 x 3 | | 14.341 ms | 2.5 ms | 5.736 |

Table 6.2: Table showing the running time of one morphological operation on an image of 320 x 240 pixels with different structuring elements

6.4 Connected Components Labelling

After the application of binary morphological operations the objects have to be labelled so that they can be used for further processing. The idea is to mark every pixel with the object number they belong to. The linear algorithm of for CCL is very slow as it traverses every pixel of the image checking for connectedness of the pixel to the neighbouring pixels. This is a very slow process. Further for 8 connectivity the algorithm has to store equivalence lists. These lists get long for large images for a large number of objects.



Figure 6.5: Output after applying CCL to an image

The regions or blobs must be uniquely labelled, in order to uniquely characterize the object pixels underlying each blob. Since there is spatial dependency at every pixel, it is not straightforward to parallelize it. Although the underlying algorithm is simple in structure, the computational load increases with image size and the number of objects, the equivalence arrays become very large and hence the processing time.

Furthermore, with all other steps being processed in parallel with high throughput, it becomes imperative to parallelize this step and to avoid it from becoming a bottleneck in the processing stream.

The parallelization of CCL is quite tricky as there as simply dividing the image into parts or working on pixels cannot be done. It may so happen that one object may get divided into two parts when we split the image for parallelization and this can cause one object to be counted as two.

The approach for parallelizing CCL belongs to the class of divide and conquer algorithms [22]. The parallelization divides the image into small parts and labels the objects in the small parts. Then in the conquer phase the image parts are stitched back to see if the two adjoining parts have the same object or not.

The steps of the parallel implementation are

- 1) Initial labelling: The image is divided into $N \times N$ small regions and each area region is worked in parallel. Each part is read from left to right top to bottom.

Let 4 – nbr denote the 4 neighbouring pixels in the directions N, NE, NW and W of the pixel p that is under consideration. The choice for these neighbours is done because these pixels would have been processed before coming to pixel p if the read order is left to right top to bottom. If p is zero then this pixel is not part of any object and no processing is done. However, if p is 1 then the labelling depends on how many number of its neighbours are 1. If p is 1 and all the neighbours are 0 then this pixel is part of a new object and a new label is assigned to it. If however, any of its 4 – nbrs is 1 then the label for that object is copied into p and all the pixels 4 – nbrs and p are marked as equivalent as they all are part of the same object.

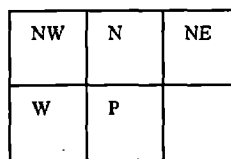


Figure 6.6: Figure showing the 4 nbrs for pixel p

We have used a data structure known as label_list that holds information about the number of objects in every small region. Note that this information is just local to every region and is not about the global object.

- 2) Update label_list: In this step we increase the label number of the object in the region counting the number of objects already seen. For example, let us take a case where region 1 and region 2 both have one object each. The information of label_list shall tell us that both regions have one object having label one. Now we will go over the structure label_list to update the information of region 2 to have label 2 as label 1 will be taken by the object to region one. This way the entire list is update for all the N x N regions that we have divided the image into. This code is run linearly.
- 3) Merge: This is part where we stitch the local information to find out objects globally. This code is not run in parallel. Let us assume that two neighbouring regions 1 and 2 have an object in common. The information till now with

label_list will be that there is one object labelled 1 in region 1 and there is one object labelled 2 in region 2. The idea to is to merge the objects in the two regions and to update label_list to reflect this.

We go over the regions left to right and top to bottom for this processing. For doing this only three things need to be compared among two regions. We need to compare the top left pixel, the first row and the first column as these are the parts of the object that can overlap with the neighbouring region. And the overlap of these regions will be good enough to conclude that whether the two objects in the two regions are same or not.

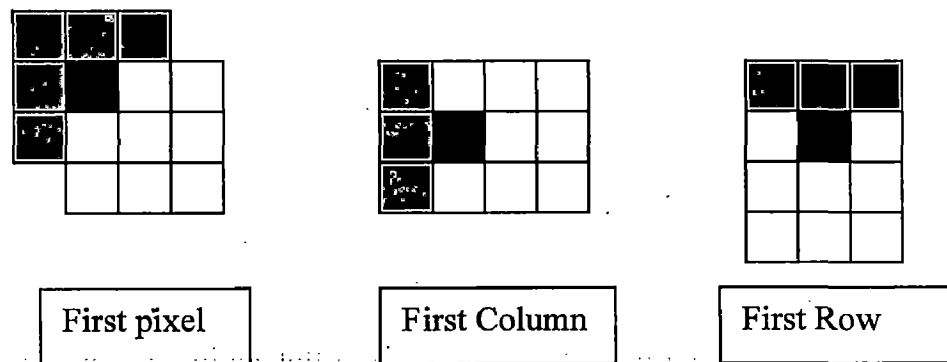


Figure 6.7: Figure showing the pixels that need to checked for overlap

Let $current_region$ be the region number for pixel p . The first pixel needs to check 5 pixels. These 5 pixels lie in 3 different regions, the regions being $current_region - 1$, $current_region - N - 1$ and $current_region - N$. Every pixel in the first row needs to be checked with 3 pixels in region $current_region - N$ and similar logic follows for the first column.

Let q be the pixel of the other region with which check is being performed if the pixel p is part of the same object. The first thing that is checked is that whether the pixel q is 1 or not. If q is zero then q is not part of any object and is not considered for resolving. If however q is 1 then the two regions may be having the same object. The algorithm for resolving the two regions is

```

Step 1: index1=label_list[region no. q][label(q)]
        index2=label_list[region i][label(p)]
        if(index1 not equal to index 2)
            perform step 2
Step 2: small_lbl=min(index1, index2)
        large_lbl=max(index1,index2)
        for k=1 to i do
            for j=1 to size of an array for Region[k].
                if ( Label_List[k][j] > Large_Lbl) then
                    Label_List[k][j] = Label_List[k][j] -1;
                else if (Label_List[k][j] = Large_Lbl) then
                    Label_List[k][j] = Small_Lbl;
            end
        end
end

```

Figure 6.8: Algorithm for resolving two neighbouring regions

At this point the task of CCL is over, however to make bounding boxes we have to figure out the boundaries till where each object extends to. This code runs the entire image pixel by pixel. The bounding box has been implemented as a structure holding the four dimensions of the bounding box.

For timings we have considered the code only that runs in parallel on CUDA and compared the same code with the linear implementation.

| Image Size | Linear Implementation | Parallel Implementation |
|-------------|-----------------------|-------------------------|
| 100 x 100 | 0.8 ms | 0.8 ms |
| 400 x 400 | 12.8 ms | 4.5 ms |
| 900 x 900 | 65.5 ms | 14.5 ms |
| 1600 x 1600 | 183.5 ms | 34 ms |

Table 6.3: Running times of CCL on linear and parallel implementation for images of varying sizes

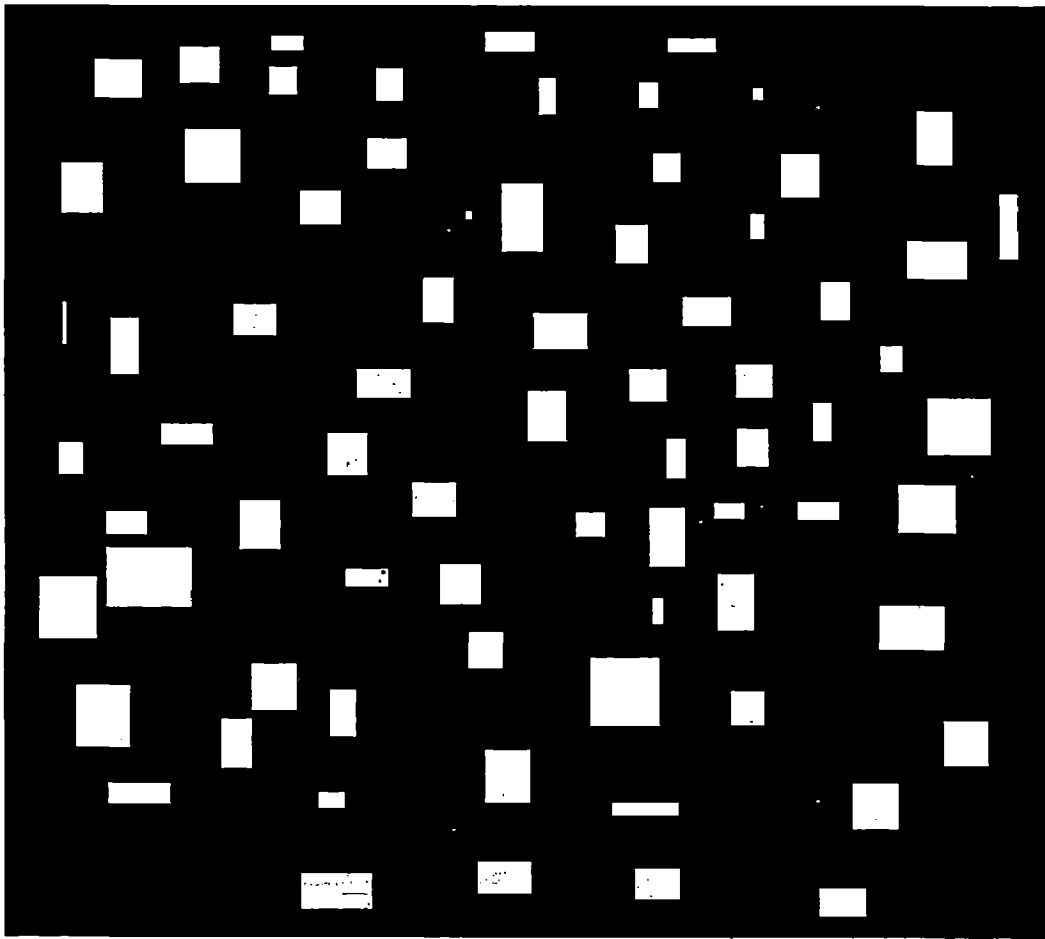


Figure 6.9: A typical input file given to the CCL parallel implementation

```
c:\Documents and Settings\Kshitz\My Documents\Visual Studio 2005\Projects\cc1_split\cc1
Number of objects 74
31 milliseconds
Label=1; Area= 3782; BB= [268 73 328 134]
Label=4; Area= 1225; BB= [408 53 456 77]
Label=6; Area= 2550; BB= [735 46 809 79]
Label=9; Area= 1800; BB= [1015 57 1089 80]
Label=12; Area= 4752; BB= [138 94 209 159]
Label=18; Area= 2021; BB= [404 107 446 153]
Label=20; Area= 2337; BB= [568 109 608 165]
Label=35; Area= 1638; BB= [817 125 842 187]
Label=37; Area= 1290; BB= [971 134 1000 176]
Label=39; Area= 320; BB= [1146 143 1161 162]
Label=49; Area= 5022; BB= [1397 182 1450 274]
Label=52; Area= 7905; BB= [276 215 360 307]
Label=56; Area= 3111; BB= [554 231 614 281]
Label=62; Area= 5481; BB= [87 274 149 360]
Label=71; Area= 2058; BB= [993 256 1034 304]
Label=73; Area= 4350; BB= [1189 257 1246 331]
Label=88; Area= 7198; BB= [761 308 821 425]
Label=97; Area= 3712; BB= [451 322 514 379]
Label=99; Area= 117; BB= [706 357 714 369]
Label=105; Area= 3108; BB= [1524 325 1551 435]
Label=113; Area= 3300; BB= [935 380 984 445]
Label=115; Area= 903; BB= [1142 360 1162 402]
Label=124; Area= 6045; BB= [1381 406 1473 470]
Label=128; Area= 3619; BB= [641 471 687 547]
Label=132; Area= 2970; BB= [1250 477 1294 542]
Label=137; Area= 360; BB= [90 515 94 586]
Label=138; Area= 3510; BB= [350 517 414 570]
Label=143; Area= 3626; BB= [1039 504 1112 552]
Label=149; Area= 4214; BB= [163 541 205 638]
Label=156; Area= 5312; BB= [809 531 891 594]
Label=173; Area= 1485; BB= [1341 588 1373 632]
Label=177; Area= 3936; BB= [539 630 620 677]
Label=180; Area= 3132; BB= [956 629 1013 682]
Label=183; Area= 3192; BB= [1120 620 1175 676]
Label=190; Area= 5220; BB= [799 666 858 752]
Label=198; Area= 9506; BB= [1412 677 1509 773]
Label=207; Area= 1914; BB= [1237 686 1265 751]
Label=212; Area= 1998; BB= [83 757 119 810]
Label=213; Area= 2765; BB= [239 724 317 758]
Label=216; Area= 4453; BB= [493 739 553 811]
Label=221; Area= 2001; BB= [1013 748 1041 816]
Label=223; Area= 3120; BB= [1121 731 1168 795]
Label=243; Area= 4012; BB= [623 824 690 882]
Label=248; Area= 7304; BB= [1368 827 1455 909]
Label=251; Area= 2418; BB= [156 875 217 913]
Label=254; Area= 5208; BB= [359 855 420 938]
Label=260; Area= 1845; BB= [873 876 917 916]
Label=262; Area= 5508; BB= [987 867 1040 968]
Label=265; Area= 1242; BB= [1086 859 1131 885]
Label=267; Area= 1953; BB= [1214 857 1276 887]
Label=293; Area= 13029; BB= [156 937 284 1037]
Label=304; Area= 9328; BB= [53 987 140 1092]
Label=312; Area= 1950; BB= [521 973 585 1002]
Label=314; Area= 4347; BB= [666 964 728 1032]
Label=320; Area= 5376; BB= [1091 981 1146 1076]
Label=335; Area= 748; BB= [991 1022 1007 1065]
Label=339; Area= 7326; BB= [1339 1035 1437 1108]
Label=355; Area= 3233; BB= [710 1080 762 1140]
Label=361; Area= 5451; BB= [378 1134 446 1212]
Label=367; Area= 12296; BB= [897 1124 1002 1239]
Label=371; Area= 8798; BB= [110 1171 192 1276]
Label=377; Area= 3200; BB= [498 1178 537 1257]
Label=383; Area= 2907; BB= [1113 1181 1163 1237]
Label=389; Area= 3864; BB= [332 1228 377 1311]
Label=403; Area= 5313; BB= [1438 1230 1506 1306]
Label=418; Area= 6120; BB= [736 1281 803 1370]
Label=424; Area= 3325; BB= [159 1338 253 1372]
Label=428; Area= 1080; BB= [480 1353 519 1379]
Label=432; Area= 5467; BB= [1298 1338 1368 1414]
Label=443; Area= 2142; BB= [930 1371 1031 1391]
Label=452; Area= 4455; BB= [724 1471 804 1525]
Label=455; Area= 6649; BB= [453 1491 561 1551]
Label=462; Area= 3710; BB= [964 1482 1033 1534]
Label=464; Area= 3456; BB= [1247 1516 1318 1563]
Press any key to continue . . .
```

Figure 6.10: Output of the CCL implementation

In this thesis we proposed a number of ways to reduce the amount of time taken for video surveillance algorithms. We used the latency performance trade-off of MOG against that of single Gaussian for speeding up the time taken for background modeling in our fusion algorithm. The trade off exploited was that multiple Gaussians take a lot more time to perform background modeling, however they are good at modeling dynamic backgrounds. Based on the type of environment our algorithm intelligently switched among the algorithms to make a choice for keeping a good balance between performance and efficiency.

We showed how support vector machines can be used to perform background modeling. Support vector machines are known to be very fast and robust tool for machine learning. We made models for every 5 x 5 block of the image thus reducing the number of models to be learnt making the algorithm even faster.

We exploited the high performance delivered by multi processors to process large amounts to data in parallel to speed up well established algorithms. We implemented EZW by exploiting the parallelism that is exhibited in the vertical, horizontal and diagonal details of the wavelet coefficients. We reduced the amount of time taken for the dominant and subordinate passes of the algorithm by performing them level wise and in parallel.

We then implemented a single Gaussian algorithm for background modeling on both STI CBE and nVidia CUDA architecture to compare the performance of CBE and CUDA. For single Gaussian background modeling CUDA architecture turned out to be faster than CBE because of the large number of cores that were available on the GPU. The higher speed up was also because of the fact that the threads for single Gaussian background modeling have less number of branch statements.

We then implemented three commonly used video surveillance algorithm on nVidia's CUDA architecture namely, GMM, Morphological operations like dilation and erosion and connected components labelling (CCL). GMM and morphological operation offered a per pixel parallelism. One thread was created for every pixel and

thus entire image was processed in parallel. CCL parallelization was done as a divide and conquer algorithm. The image was divided into small tiles and each of the tiles was labelled in parallel. After this processing the tiles were merged to ensure that if an object was part of more than one tile then that object is considered only once. This was followed by marking of bounding boxes across all the objects.

A good future work could be fusion of information from multiple cameras to get more information about the environment. This sort of information can be used to aid in suppression of shadows. Parallel implementation of various machine learning algorithms can be done and be applied to background modeling to be performed in real time. Another work that can be done is to perform object detection in the compressed domain. This work should have parallel implementations for wavelet transform, EZW and then the objects could be detected in the compressed domain. This can reduce the amount of time taken as the coefficient need not be decoded. Use of shared memory and other parallel techniques can be used to improve the speed up of the codes that have been implemented.

References

- [1] Harville, M.; Gordon, G.; Woodfill, J., "Adaptive video background modeling using colour and depth," Proceedings of International Conference on Image Processing, vol.3, pp.90-93, October 7-10 2001, Thessaloniki, Greece.
- [2] Maddalena, L.; Petrosino, A., "A Self-Organizing Approach to Background Subtraction for Visual Surveillance Applications," IEEE Transactions on Image Processing, vol.17, no.7, pp.1168-1177, July 2008
- [3] Alexandre F., Gerald M., "Adaptive colour background modeling for real-time segmentation of video streams", Proceedings of International Conference on Imaging Science, System and Technology, pp.227-232, June 28, 1999, Las Vegas, Nevada, USA.
- [4] Stauffer, C.; Grimson, W.E.L., "Adaptive background mixture models for real-time tracking," IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol.2, pp.-252, 23 – 25 June 1999, Ft. Collins, CO, USA.
- [5] Toyama, K.; Krumm, J.; Brumitt, B.; Meyers, B., "Wallflower: principles and practice of background maintenance". The Proceedings of the Seventh IEEE International Conference on Computer Vision, vol.1, pp.255-261, 20-25 September, 1999, Kerkyra, Corfu, Greece.
- [6] Javed, O.; Shafique, K.; Shah, M., "A hierarchical approach to robust background subtraction using colour and gradient information," Workshop on Motion and Video Computing, pp. 22-27, 5-6 Dec. 2002, Orlando, Florida.
- [7] Jain, V.; Kimia, B.B.; Mundy, J.L., "Background Modeling Based on Subpixel Edges," IEEE International Conference on Image Processing, vol.6, pp.VI -321-VI -324, Sept. 16 2007-Oct. 19 2007, San Antonio, Texas, USA.
- [8] Richardson E. Iain "H.264 and MPEG 4 Video Compression" Wiley Publications.

[9] Indupalli, S.; Ali, M.A.; Boufama, B., "A Novel Clustering-Based Method for Adaptive Background Segmentation," The 3rd Canadian Conference on Computer and Robot Vision, pp. 37-37, 07-09 June 2006, Quebec City, Canada.

[10]

http://upload.wikimedia.org/wikipedia/commons/7/74/Normal_Distribution_PDF.svg,

Last accessed on 29th May 2009

[11] Shimada, A., Arita, D., Taniguchi, R., "Dynamic Control of Adaptive Mixture-of-Gaussians Background Model," IEEE International Conference on Video and Signal Based Surveillance, 2006, pp.5-5, Nov. 2006, Sydney, Australia.

[12] Kim H., Suryanto, Kim D., Zhang D., Ko S., "Fast object detection for video surveillance", Proceedings of International Technical Conference on Circuits/Systems, Computers and Communications, pp. 709 – 712, July 9 2008, Shimonoseki City, Japan.

[13] DD_Tools Homepage http://ict.ewi.tudelft.nl/~davidt/dd_tools.html Last accessed on 29th May 2009

[14] PrTools Homepage <http://www.prttools.org/> Last accessed on 29th May 2009

[15] "Caviar: Context aware vision using image-based active recognition," <http://homepages.inf.ed.ac.uk/rbf/CAVIAR/>.

[16] NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide. NVIDIA Corporation, Jan 2007.

[17] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," IEEE Transactions on. Signal Processing, vol. 41, no. 12, pp. 3445-3462, Dec. 1993.

[18] S. Kandadai and C. D. Creusure, "An experimental study of object detection in the wavelet domain", Conference record of the thirty-seventh Asilomar conference on Signals, Systems and Computers, vol. 2, pp 1620-1623, Nov 2003.

[19] H. N. Cheung, L. Ang and K. Eshraghian, "Parallel architecture for the implementation of the embedded zerotree wavelet algorithm", 5th Australasian conference on computer architecture 2000, pp 3 – 8, 31st Jan – 3rd Feb 2000, Canberra, Australia.

[20] L. Ang, H. N. Cheung and K. Eshraghian, "EZW algorithm using depth-first representation of the wavelet zerotree", Proceedings of the fifth international symposium on signal processing and its applications 1999, vol 1, pp 75-78, 22nd Aug – 25th Aug 1999, Brisbane, Australia.

[21] "Normal Distribution – Wikipedia, the free encyclopaedia"
http://en.wikipedia.org/wiki/Normal_distribution Last Accessed 29th May 2009

[22] J. M. Park, C. G. Looney, H. C. Chen, "Fast Connected Component Labelling Algorithm Using A Divide and Conquer Technique." Technical report, 2000.
<http://cs.ua.edu/research/TechnicalReports/TR-2000-04.pdf> Last Accessed 29th May 2009

Publication

1. K. Gupta and A. Mittal, "Parallelization of EZW on GP GPU programming paradigm", 3rd International Conference on Information Processing 2009. (Accepted May 2009)