HIGH PERFORMANCE AUTOMATIC TARGET RECOGNITION

Misiker Tadesse

Advisor

Dr. Eneyew Adugna

A thesis submitted to

The Department of Electrical and computer Engineering

Presented in partial fulfillment of the requirement for the degree of

Masters of Science in Microelectronics Engineering

Addis Ababa University

Addis Ababa Institute of Technology

Addis Ababa, Ethiopia

October 2012

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

HIGH PERFORMANCE AUTOMATIC TARGET RECOGNITION

By

Misiker Tadesse

ADDIS ABABA INSTITUTE OF TECHNOLOGY

APPROVAL BY BOARD OF EXAMINERS

_____	_____
Chairman, Dept. of Graduate committee	Signature
Dr. Eneyew Adugna	_____
Advisor Signature	
_____	_____
Internal Examiner	Signature
_____	_____
External Examiner	Signature

DECLARATION

I, the undersigned, declare that this thesis work is my original work, has not been presented for a degree in this or any other universities, and all sources of materials used for the thesis work have been fully acknowledged.

Misiker Tadesse _____

Name signature

Place: Addis Ababa

_____

Date of submission

This thesis has been submitted for examination with my approval as a university advisor.

Dr. Eneyew Adugna _____

Advisor's name Signature

ABSTRACT

Designing a vision system, which was motivated by that of the human eye, has been done since the introduction of digital computing devices. Due to its abundant applications, this area is a hot topic in the industry. Its computational complexity hinders it from the required accuracy and flexibility achievable by these systems.

The evolution of processing technology has given to the birth of parallel processing units such as Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA). Using these parallel processing units Data Intensive applications can be done in parallel to get a performance speedup with lower implementation cost, unlike Application Specific Integrated Circuits (ASICs).

In this thesis, a real-time Automatic Target Recognition system having the ability of detection, identification and tracking of pre-specified targets even with bad visual artifacts that could originate from lightning conditions, environmental conditions or other noise generating circumstances is developed. This thesis work investigates an improved combination of detection and recognition algorithms to reach a better solution to the problem of target detection and recognition while aiming to fit the design of a minimal system. For each algorithm involved the theoretical background is investigated and a detailed description of the realization is given followed by an analysis of both achievements and short comings of the design.

A speed up of 2.4 is achieved, by using a non-preeminent GPU, over brute force implementations.

ACKNOWLEDGMENTS

First and for most I would like to thank my advisor Dr. Eneyew Adugna for his unspeakable support and encouragement. His dedication and guidance made this thesis possible. I have perceived unforgettable values under his supervision. I am also grateful to the Department of Electrical and Computer Engineering for giving me this opportunity.

The constant encouragement and support of my friends especially Biniam, Bisrat, Abraham and Daniel was invaluable for the success of this thesis. I would also like to thank people from OpenCV yahoo group for their quick and resourceful responses for my technical inquiries.

I would also thank my beloved family especially Mitia, Etete, Tadye and Miye for their indescribable gifts and beliefs in me. The love and support of my family cannot be paid back at all. Finally my thanks go to all those who were by my side in doing this thesis.

Contents

List of Figures

List of Tables

List of Acronyms

GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
ATR	Automatic Target Recognition
CPU	Central Processing Unit
HSV	Hue Saturation Value
HSB	Hue Saturation Brightness
ILP	Instruction Level Parallelism
TLP	Task Level parallelism
DLP	Data Level Parallelism
ASIC	Application Specific Integrated Circuit
HDL	Hardware Description Language
FLOPS	Floating Point Instruction Per Second
GFLOPS	Giga FLOPS
SDK	Software Development Kit
CUDA	Compute Unified Device Architecture
CPI	Clocks Per Instruction
SIMD	Single Instruction Multiple Data
HD	High Definition
SLM	Spatial Light Modulator
SAR	Synthetic Aperture Radar
UAV	Unmanned Aerial Vehicle
ROI	Region of Interest
SM	Streaming Multi-processors
PC	Program Counter
SIMT	Single Instruction Multiple Thread
QVGA	Quarter Video Graphics Array
HVGA	Half Video Graphics Array
VGA	Video Graphics Array

SVGA	Super Video Graphics Array
XGA	Extended Graphics Array
XGA+	Extended Graphics Array plus
SXGA	Super Extended Graphics Array
SXGA+	Super Extended Graphics Array plus
UXGA	Ultra Extended Graphics Array
QXGA	Quad Extended Graphics Array

CHAPTER 1

INTRODUCTION

1.1 Introduction

With the recent advent of stream computing many multimedia, image and signal processing applications have fallen in this category. As their name suggests these applications operate on streams of data and hence are typically characterized by their high computational performance requirements. Automatic Target Recognition (ATR), being a real time signal processing application, essentially entails to this high performance requirement.

Automatic Target Recognition can be referred to as using of computational processing power of digital computers for detecting and recognizing targets in a sensor data. The sensor data could be images from Infrared camera, Synthetic Aperture Radar, Television camera or from non-image sensor data. ATR has evolved to be a very important part of intelligent systems both for military and civil applications. In military, ATR has become part of military Jets by increasing accuracy and efficiency of attacks, while minimizing the collateral destruction on non-targets. In civilian application it is being used as part of intelligent surveillance system, robotic vision systems, as well as for landmark recognition systems.

Various improvements have been done and are being done from various vantage points towards automatic target recognition. Signal processing algorithms, processor technology, sensor technology improvements have a great impact on the evolution of ATR. Processor improvement has brought with it parallel architectures which are suited

to the application of signal processing for applications which are data intensive for operations with very less sequential flow. ATR falling in this category can take advantage of the parallelism in the processors.

1.2. Statement of the problem

Automatic target recognition is an important area of research under Computer Vision where a computer system is made to mimic a human visual system to recognize and track object(s) of interest(s). ATR finds its applications in many industries such as traffic control, automobile, defense, and entertainment. A successful teal-time ATR algorithm must meet performance (timing) and accuracy requirements imposed on it for accurately tracking objects of concern irrespective of the external environmental conditions, surrounding the target.

Noise, occlusion and change in lighting intensity are prominent challenges that are obstructive to ATR algorithm in recognizing and tracking the target. Intermediate stages in ATR such as noise cancellation operations and intensive recognition operations are time consuming and tradeoffs to speed.

To compensate for the speed drop due to the added features on the algorithm, a parallel computing chip particularly GPU is used to assist ATR algorithm to accelerate the process of recognition and related intensive computations. The ATR algorithm has many phases, not every operation in the ATR algorithm is able to be executed in parallel. The

essence is to redesign serial implementation of the algorithm to run on GPU to achieve a better speedup of the whole system.

1.3. Objective

a. General Objective

The main objective of this thesis is to develop and implement a high performance automatic target recognition algorithm and evaluate its performance against previous works.

b. Specific Objectives

The following are specific objectives of the thesis:

- Explore the subject of automatic target recognition and its performance requirements and fill the gap in the performance requirement.
- Design and implement a target template which minimizes the timing constraint through minimizing area covered by the template matching.
- Implement the template matching in parallel on GPU to achieve a performance speed up.
- Evaluate the algorithms performance and suggest design improvements.

1.4. Scope of the thesis

The main intention of this thesis is to design and implement a real time automatic target recognition system that is immune to occlusion and simple Gaussian noises which could come from uncertainties in the camera or the environment. Other type of noises such as image ghosts and strong shadows are out of the scope of this thesis.

In the detection stage, both shape and color based segmentation are used. Shape analysis using background subtraction and consecutive morphological operations, to avoid noises, candidates of potential target regions to be identified/recognized are selected. The color based segmentation uses the color range of the target template to filter out non-target parts from the frame being considered. The above two segmentation algorithms results are merged to achieve better segmentation accuracy without affecting the performance. In the recognition step, identifying the potential targets detected from the segmentation step is performed using Template Matching approach.

The algorithm can be applicable to the video stream while capturing from camera. Hence, the implementation works as fast as the video stream making it a real time applicable algorithm. The C++ programming language is used for this work, and because of its processing speed, Open Computer Vision Library (OpenCV) framework is used for the implementation. On the other hand, CvBlob framework is used for blob detection in the segmentation phase.

1.5. Thesis Contributions

In this thesis, a combined segmentation algorithm is developed which uses a composite template type. Evaluation of this template has been made. This technique has been suggested by the OpenCV developer's network in their discussion group [30].

A GPU based template matching algorithm has been implemented to get a speed up of the recognition phase of ATR from conventional methods using CPU implementation. The recognition phase the most computational bottle neck of the whole system, its performance improvement has led to a speed up of the whole system.

1.6. Outline of the Thesis

This chapter, chapter 1, introduces the area of target recognition and the target recognition algorithm that is used in this thesis. The remaining chapters of this thesis are organized as follows. In chapter 2 the background of automatic target recognition and Graphics Processing Unit (GPU) are thoroughly discussed followed by discussion of previous works done in the area. In chapter 3, the design of the proposed algorithm is discussed. Chapter 4 discusses the implementation details and the evaluation of the proposed system. Chapter 5 discusses the conclusion and recommended future works.

CHAPTER 2

LITERATURE REVIEW

2.1. BACKGROUND THEORY

In this section terms and concepts used in this thesis are thoroughly discussed. Particularly terms and concepts in the area of ATR and parallel processing for speed up are given comprehensive emphasis.

2.1.1. Automatic Target Recognition

The basic principle behind ATR is detecting and recognizing specific items of interest (or targets) in an image gathered from complex background (or clutter) by a non-flawless sensor, which inevitably introduces noise in the gathered signal. The context of clutter, target and noise depend on the particular application at hand. For instance target could be a moving object in general or a particular type of object in some region. The former is called classification, while the later is called identification or recognition. Clutter refers objects which are contained in the gathered image and cover most of the area as compared to the target, but are not of interest to the particular application. Noise is an unwanted signal which is introduced by camera imperfections, environment or round-off errors during computation.

Main challenges in ATR can be categorized in to: elimination of noise from a noisy signal gathered or extracting targets from a complex surrounding. ATR problems generally include a front-end target detection (or segmentation) stage [10]. The main aim of the detection stage is to minimize the amount of data on which the process of target recognition is done. This means detection refers to finding a probable target of interest

which needs to be verified in the recognition stage. The target detection stage minimizes the number of computation required while making sure not to miss targets of interest. The following figure shows the elimination of background clutter and non-target clutter in an ATR pipeline.

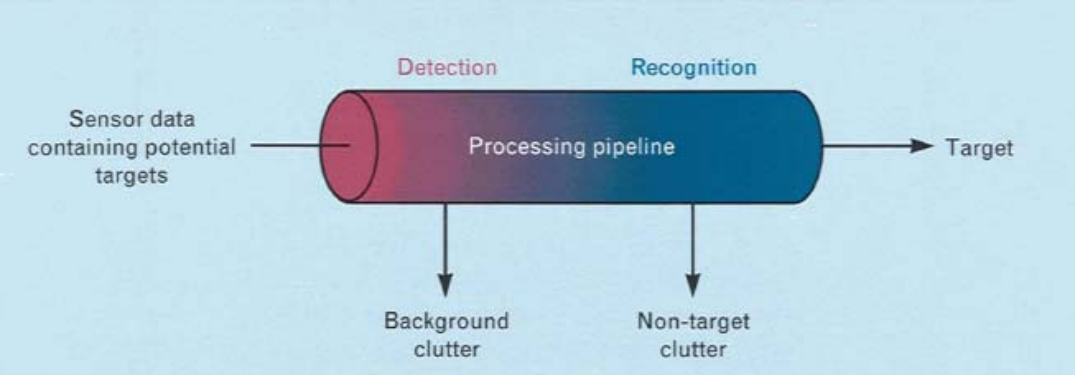

Figure 2.1: Conceptual data flow in automatic target recognition (ATR) systems [10]

Generally the ATR algorithm can be divided in to five major steps: Sensing, Detection/ Segmentation, Feature Extraction, Target Recognition, and Post-Recognition processing. Sensing stage constitutes collecting data from sensor, for this work it could be either capturing an image from video stream of camera, or capturing a frame from a video input data. The detection/segmentation stage on the other hand aims to isolate interesting parts, parts of the image which could be targets, for further processing. The result of this stage contains a frame with binary value, setting 0 and 1for detected and undetected pixels of the image respectively. Feature Extraction is the stage which accepts the binary data from the detection stage and returns the regions which are going to be passed for the target recognition stage of the system. The recognition stage involves taking the regions of interest and finding out whether the target is present in the region based on predefined

features of the target. The Post Processing stage, considering the result of the recognition stage will finally issue the presence and absence as well as tracking of the target.

Segmentation

Segmentation is the first step in any image analysis and target recognition application. It is most essential and most difficult task and determines the performance of the whole system. It is basically dividing the image into different regions which are heterogeneous, and any two regions in the same category are homogeneous. The formal definition of segmentation is given in [25]:

If $Homogineity()$ is a homogeneity predicate defined on a group of connected pixels, then segmentation is a partition of the set F in to connected subset or regions $\{S_1, S_2, \dots, S_n\}$, such that:

$$\bigcup_{i=1}^{n} S_i = F \,, with\, S_i \cap S_j = \emptyset \;\; (i \neq j) \tag{2.1}$$

The homogeneity predicate $Homogeneity(S_i) = True$ for all segmented homogeneous region, S_i , and $Homogeneity(S_i \cup S_j) = False$, where $i \neq j$, and S_i and S_j are adjacent regions.

For target recognition applications the subset/regions are 2, the target and the background clutter.Segmentation can be of two types, depending on the feature of the image to be detected: Color based Segmentation or Shape based segmentation such as Background subtraction segmentation.

Color Segmentation

Almost all perceivable colors can be produced by a combination of three different colored lights. Accuracy of the measurement of the three color components is the basis of color segmentation and color matching. By using typical colors from the spectrum, components required to model any color of known color combination can be represented. The three components are often chosen in such a way that two of them are chosen from the ends, and one from the middle of the visible spectrum. Different colors can be produced by varying the amounts of the three components, once they are set. Usually the three values to represent any color are known as tri-stimulus values of the color represented [28]. There are lots of color space representations, to name a few:

RGB

These color space is based on proper combination three colors sampled at three different ranges of wavelengths, Blue in the short range (430nm - 480nm), Green in the medium range (520-570 nm) and Red in the long range (630-700 nm).

Given the spectral energy distribution of the light E_λ then any of the color components can be represented by integrating the individual color components' matching function, shown in Figure 2.2, with the spectral distribution.

$$\boldsymbol{R} = \int \boldsymbol{E_\lambda r_\lambda d\lambda} \tag{2.2}$$

$$\boldsymbol{G} = \int \boldsymbol{E_\lambda g_\lambda d\lambda} \tag{2.3}$$

$$\boldsymbol{B} = \int \boldsymbol{E_\lambda b_\lambda d\lambda} \tag{2.4}$$

Where $\mathbf{E_\lambda}$ is energy distribution of the light, $\boldsymbol{r_\lambda}, \boldsymbol{g_\lambda}$ and $\boldsymbol{b_\lambda}$ are the wavelength ($\boldsymbol{\lambda}$) distributions of red, green and blue respectively

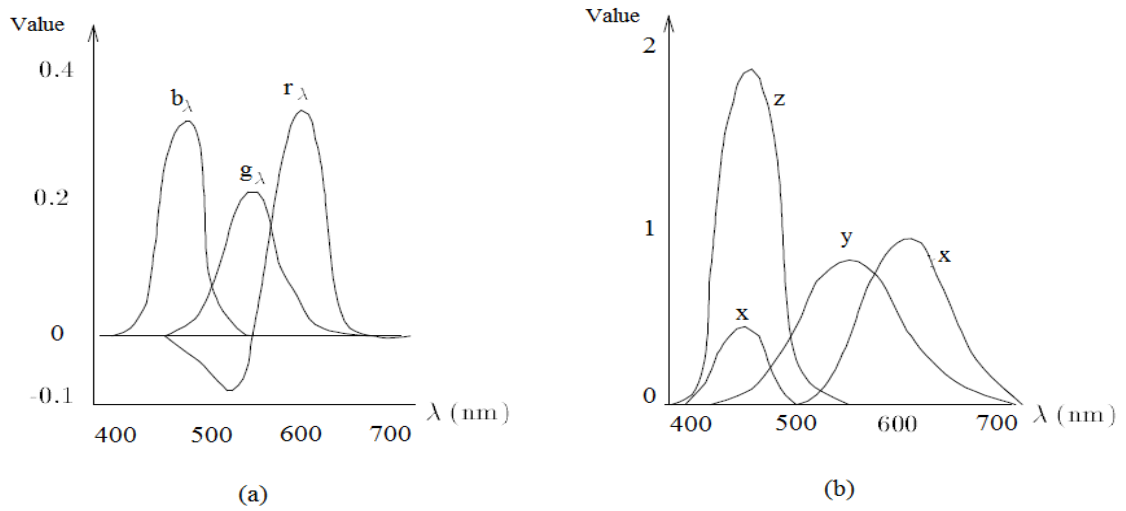

Figure 2.2: a) Color matching function in RGB b) Color matching function in XYZ [27]

HSV

HSV which stands for hue, saturation, and value is the most common cylindrical coordinate representations of points in an RGB color model. HSV representations rearrange the geometry of RGB in an attempt to be more intuitive and perceptually relevant than the Cartesian (cube) representation. It was developed in the 1970s for computer graphics applications. It is also often known as HSB (B for brightness).

Hue (H) is the color part of the visual perception, basically the stimulus on retina to light within a narrow range of wavelength. Saturation represents the relative whiteness of the perceived stimuli for a given Hue value. V is the intensity value, aka brightness, is the lightness of the color which is in the range of dim to bright. The Value/Brightness component is the one responsible for variation like orange and brown, or black, gray and brown. The Value (V) is directly related the intensity of light available. So, this value is the value which varies for Indoor/Outdoor data for this thesis.

XYZ

XYZ is a color space which has positive color matching functions across the entire visible spectrum was developed by International Commission on Illumination (CIE). The components X, Y, and Z are totally imaginary and have no correlation with the value of wavelength in the visible spectrum. It was developed in such a way that all color matching and their corresponding components are performed having positive coefficients. There have been lots of published researches for this color space in CIE conferences.

Morphological Operations in Image Segmentation

Binary morphological operations consist of a set of operations used to study and change geometric properties in binary images. It is basically used for finding, enhancing and/or removing certain geometric features in an image. For instance, it could be used for detecting contours of objects in a frame, closing small gaps in an object, etc. The most important operations for this paper are dilation, erosion and the median filter. All of these operations are based on operations on two or more sets [34]. One of the sets is the image on which we are performing the morphological operation and particularly the ON pixels in the image; the other sets are referred to as structuring elements using which the operation is performed. Basically, a structuring element can be any size and have its origin at any location within the predetermined size. Elements in the structuring elements are represented by ordered pairs in a two dimensional Euclidean space, which is an ordered pair indicates the position relative to the origin of the image.

Dilation

Dilation, $\oplus$,basically combines a given image with the structuring element set using Minkowski addition pair by pair on each element of the set with all elements of the other [34]. Dilation operation can be expressed mathematically:

$$A \oplus B \leftarrow \{c \in E^2 : c = a + b, a \in A, b \in B \tag{2.5}$$

Dilation is often used to fill small gaps in images, or for object growing operations [34]. It can also be used in conjunction with Erosion and Median Filters for image de-noising applications.

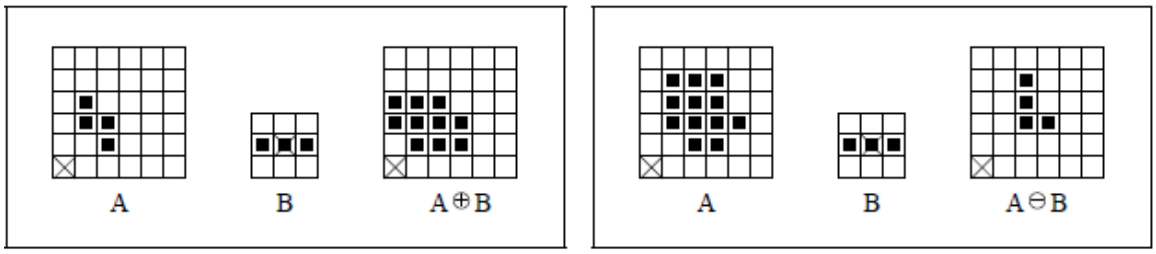

Figure 2.3: Morphology Operations: Dilation (Left) and Erosion (Right)

Erosion

Erosion,$\ominus$, is the inverse of dilation. It relates the image and structuring element sets by an operation analogous to subtraction [34]. Erosion operation can be described mathematically as:

$$A \ominus B \leftarrow \{c \in E^2 : c + b \in A, for\ each\ b \in B \tag{2.6}$$

After comparing the structuring element is with every pixel in the image if all ON pixels in the structuring element are also ON in the image, then the pixel being considered, the

one at the origin of the structuring element resides on, is ON in the output image. Graphical examples of erosion and dilation are shown in Figure 2.3.

2.1.2. Parallelism in computation

Several types of parallelism are performed in computing; based on the level of parallelism; it can be classified in to three major groups:

- Instruction Level parallelism(ILP)
- Task Level parallelism(TLP)
- Data-Level parallelism(DLP)

Instruction-level parallelism

Any computational application includes a set of instructions which are executed by a processing unit one after the other. Using principle of ILP reordering and grouping of the instructions is performed in such a way that the result is not changed and the groups can be executed in parallel.

Modern processors have multi stage instruction pipelines so that every stage in the pipeline performs different action on the instruction. An N stage pipeline can have up to N different instructions at different stages in the pipeline.

ILP achieves the parallelism by rearranging the instructions in such a way that no two instructions have data dependencies in a group of instructions; So that the group of data can be in different stages of the pipeline at the same time.

Task Level parallelism

TLP is performed in multi core processor systems in such a way that every processor performs different threads. Each processor executes a different thread or process on the same or different data. The threads may execute the same or different code. Usually threads communicate with each other by passing data with each other.

Data Level parallelism

DLP is said to be achieved when every processor performs the same task on different data. In such situations, either a single thread controls a particular operation on all pieces of data or different threads control all the operations on a single data.

For example if we take matrix addition of two matrices of the size NxN, to demonstrate DLP, the matrix can divide the matrix in N_p parts such that $1 \leq N_p \leq N^2$; where N_p is the number of processors available. For the N x N data we have, the processors perform the addition on their part of the matrices. Theoretically, assuming unlimited memory bandwidth, the matrix addition operation should be in this case N_p times faster than performed on a single processor.

Data Dependency

While implementing parallel algorithms, the most essential part to consider is data dependencies between the operations. The fundamental principle is that an execution of a given program can't run faster than the longest data dependent calculations, aka critical path. Data dependent computations in the critical path must be executed in such a way that there order is not violated in the chronology of the computation.

The instruction in an algorithm or part of an algorithm can be performed in parallel, if there is no data dependency between each other, as shown in Figure 2.4.

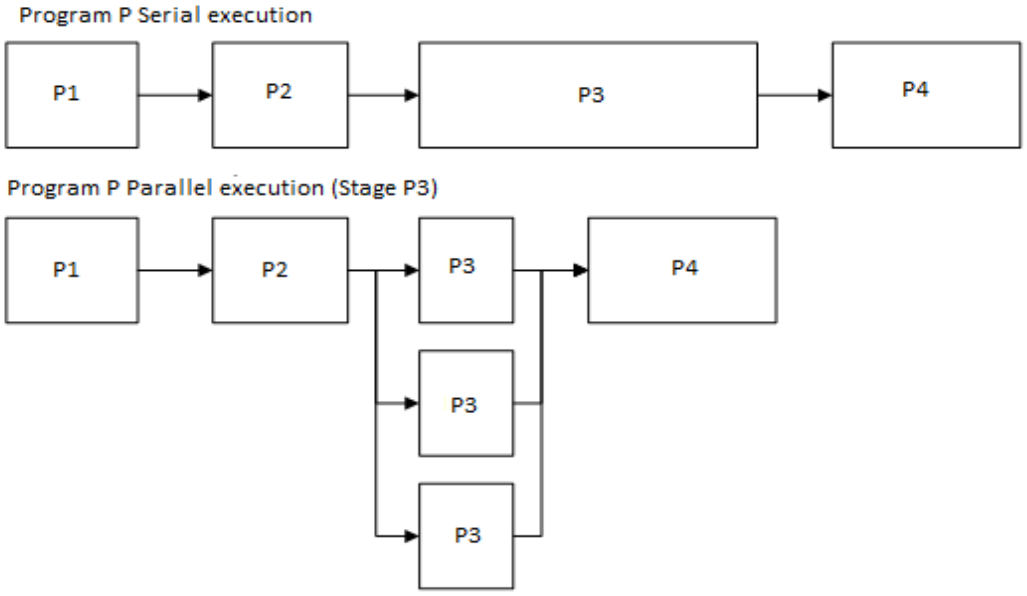

Figure 2.4 : Performance improvement executing part of algorithm in parallel.

Parallelism Constraints

The speed up achieved by parallelism of computation is linear in theory, i.e. for N processors the computation time should be $1/N^{th}$ of the serial computation. In practice the computational speed up achieved converges beyond some optimal number of processors, as shown in Figure 2.5.

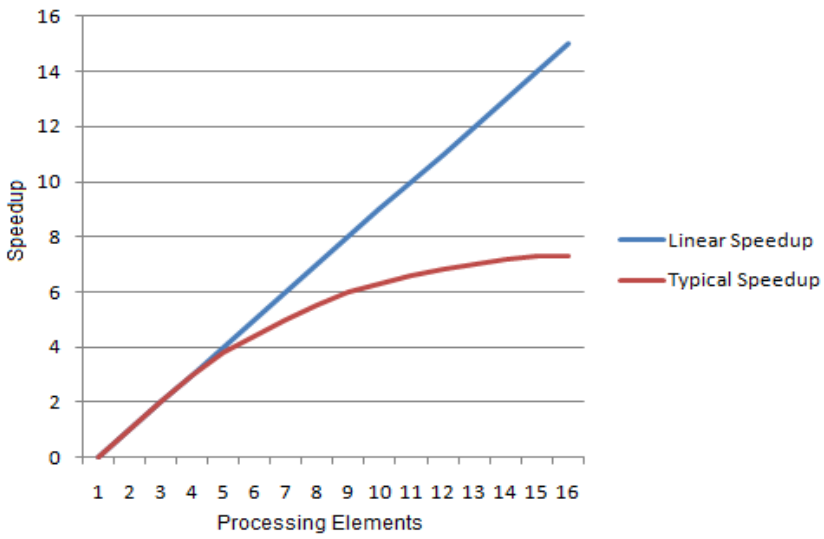

Figure 2.5: Theoretical vs. practical speed up, [20]

According to [20], this characteristic is merely due to hardware constraints of the processor, such as the global memory bandwidth, Input Output speed, the throughput constraint of the peripheral bus.

Amdahl's law is usually used to determine the achievable speed up of an algorithm using parallel computation units. Most algorithms are composed of instructions which are parallelizable parts and non parallelizable parts. Amdahl's law states that the achievable speed up of an algorithm is constrained by no-parallelizable part of the algorithm. [20] Their relationship can be described by the following equation, Amdahl's law, [20].

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.7}$$

Where: S is the achievable Speed up

N is the number of processing elements

P the ratio of the parallelizable part

According to Amdahl's law, if the non-parallelizable part is 50% of the total algorithm then a speedup of more than 2 times cannot be achieved, no matter how many processors are there to process. Theoretically the computational complexity of an algorithm could be as fast as minimal runtime of the slowest parts.

Amdahl's law assumes that the size of the sequential section is independent of the number of processors. An approach which does not exploit the computing power that becomes available as the number of processors increases. To deal with this, Gustafson's [21] proposed that larger problems can be solved if more parallel equipment is available. The key assumption of Gustafson and Barsis [21] is that the total amount of work to be done in parallel varies linearly with the number of processors. Then practical implication is of the single processor being more capable than the single processing assignment to be executed in parallel with (typically similar) other assignments. This implies that b, the per-process parallel time, should be held fixed as N, number of processors, is varied. The corresponding time for sequential processing is:

$$a + N.b$$

The scaled speed up is achieved is:

$$S = \frac{a + N.b}{a + b}$$

But $P = \frac{b}{a+b}$, parallel fraction of the execution time

$$S = N - (1 - P) * (N - 1) \tag{2.8}$$

Where N is the Number of Processors,

S is the speedup, and

P is parallelizable part of the process.

Through Gustafson's law, Amdahl's law characteristics can be modified as shown in Figure 2.6.

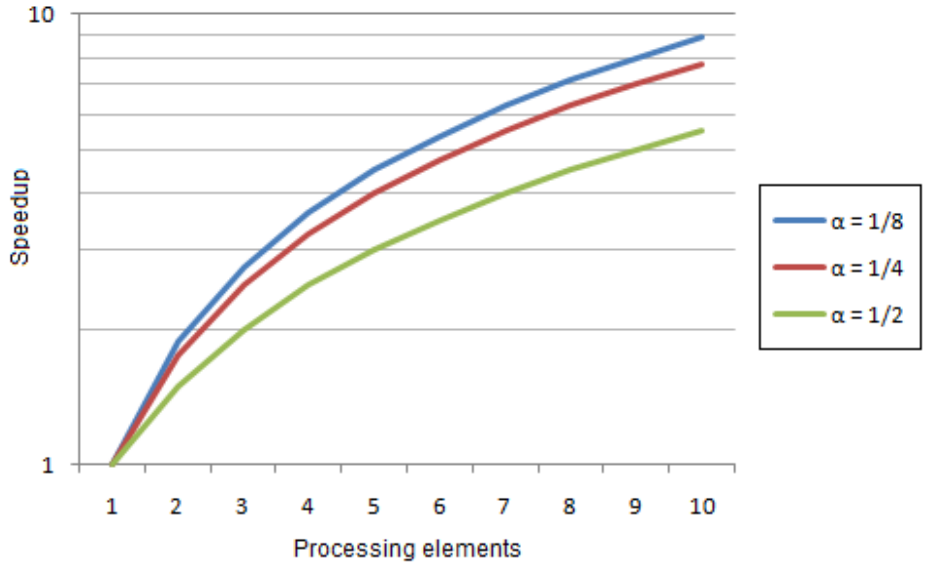

Figure 2.6: Various speedups with different α (non-parallelizable part)

One approach for accelerating computationally intensive tasks is to transfer some or all of the computations to dedicated hardware, such as a custom Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) co-processor. FPGAs are essentially field programmable logic devices where any logic function can be implemented in parallel by representing the design with a Hardware Description Language (HDL), such as VHDL or Verilog and synthesizing it on the device. This FPGA approach was very successfully applied to template matching speed up in [8] and to the acceleration of various signal processing operations [10]. In spites of the achievable performance gains, FPGAs are often expensive and involve high complexity both in terms of interfacing the FPGA device with the host PC, peripheral storage, and input devices required for an operational system and expressing the problem in an inherently parallel HDL. Apart from that, FPGAs implementation lack the flexibility

requirement set by some applications as its most parameters are fixed at the compile time of the HDL code.

Graphics Processing Units is a reduced cost speed up alternative to FPGAs. Driven by the ever increasing market demand for realistic3D games, GPU devices have evolved into highly parallel, multithreaded, multi-core processing units with enormous computational power.

In terms of Floating Point Instruction Per Second (FLOPS) or Giga FLOPS (GFLOPs), modern commodity GPUs are overshadowing CPU performance. The computational capability of GPUs, overall computational frequency, has been growing at average yearly rate of 1.7 (pixels/second) to 2.3 (vertices/second), which is a significant margin over the average yearly rate of roughly 1.4(pixels/second) for CPU performance [8].

2.1.3. Graphics Processor Unit (GPU) Background

Due to their parallel nature GPUs are more efficient to solve scientific and engineering problems with very large datasets. Previous generation of GPUs were very complicated to be used as general purpose programming units for general purpose computations. For solving algorithms on GPUs a sequence of translation and rotation had to be used. Accessing of memory was also as complicated.

In recent days, most of the graphics card manufacturers, mainly NVIDIA, came up with a software development kit (SDK) that allows software developers to exploit the power of GPU's for general purpose computations. The use of general purpose computation on GPUs for general purpose computation is through random access memory as NVIDIA's

Compute Unified Device Architecture (CUDA) rather than formulating problems like the olden days graphics paradigm.

GPU and CPU Compared

CPUs are very much optimized for sequential computation of programs. Generally, CPUs do computation on single unit of data for a given cycle. Most of the devices used are devoted for data cache to decrease the overall clocks per instruction (CPI) and flow control units, which are the main building blocks of CPUs. Since the operations are executed in sequential fashion, most of the devices are used to optimize the execution time of a single instruction. The cache memory is used to pre-fetch data by employing spatial and temporal locality within the running program. And the flow control unit is used to do execute branching operation and to do branch predictions, as in Tomasulo's algorithm for current CPUs. Most of the die area being occupied by the above units, very small portion of the total die area is dedicated for Arithmetic and Logic units (ALU), since only a single instruction is executed at a time. Even on modern CPUs, which have multiple cores on a single chip, the trend is to simply duplicate the above architecture on the chip.

As opposed to CPUs, GPUs take a completely different approach to computation of programs. If we start with their architecture, rather than having most portion of the area for data caching and complex control units, GPUs employ most of the area for data processing units as shown in Figure 2.7. This makes GPUs suitable for computation of programs which have very large number of threads. Hence, CPUs can only handle very small number of threads like four, while GPUs can handle a large number of threads, in

order of thousands. The drawback with GPU execution is that, if we take a single thread its performance is very much degraded when compared to its CPU equivalent. The performance gain of GPU computation can only be achieved for computation of parallelized operations only; otherwise the CPU is the most efficient choice.

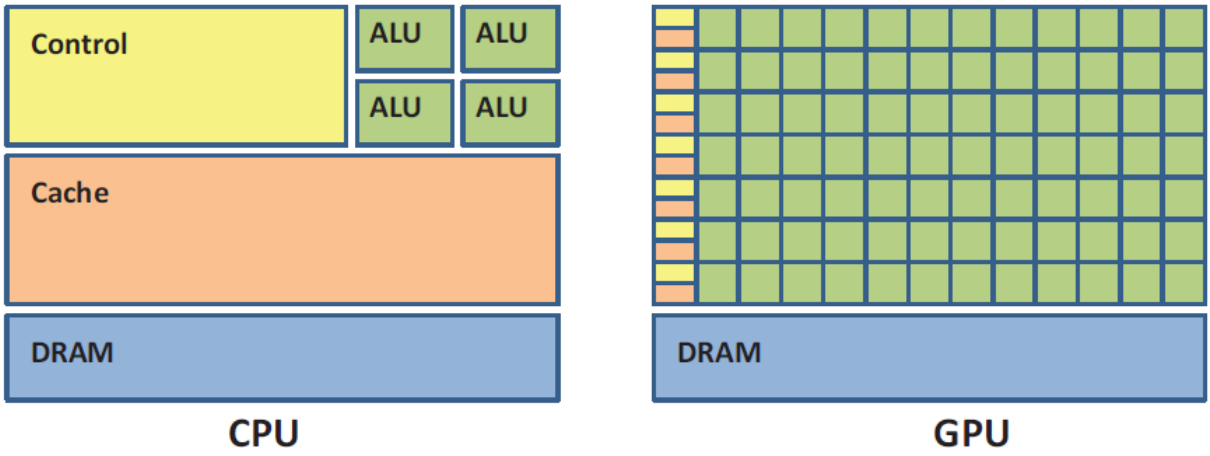

Figure 2.7: CPU Vs. GPU amounts of transistors for cache, control logic and data processing [13].

The current trend in the graphics industry is using of high definition (HD) 3D technology for graphics application such as games, motion pictures etc, which requires computation of data-parallel operations—a single task that operates identically and independently on a set of data[12]. Data-parallel operation aka Single Instruction Multiple Data (SIMD), usually do not need the output of one operation as an input to the other, which is suitable for GPU computation. Due to the high speed computational demand of HD 3D graphics, for real-time application the programmable GPUs has evolved into a highly parallel, multithreaded, multi-core processor with tremendous computational capacity as shown on Figure 2.8. The bus and memory bandwidth which acts as a bridge between the host (CPU) and the device (GPU) is also greatly improving through time, Figure 2.9.

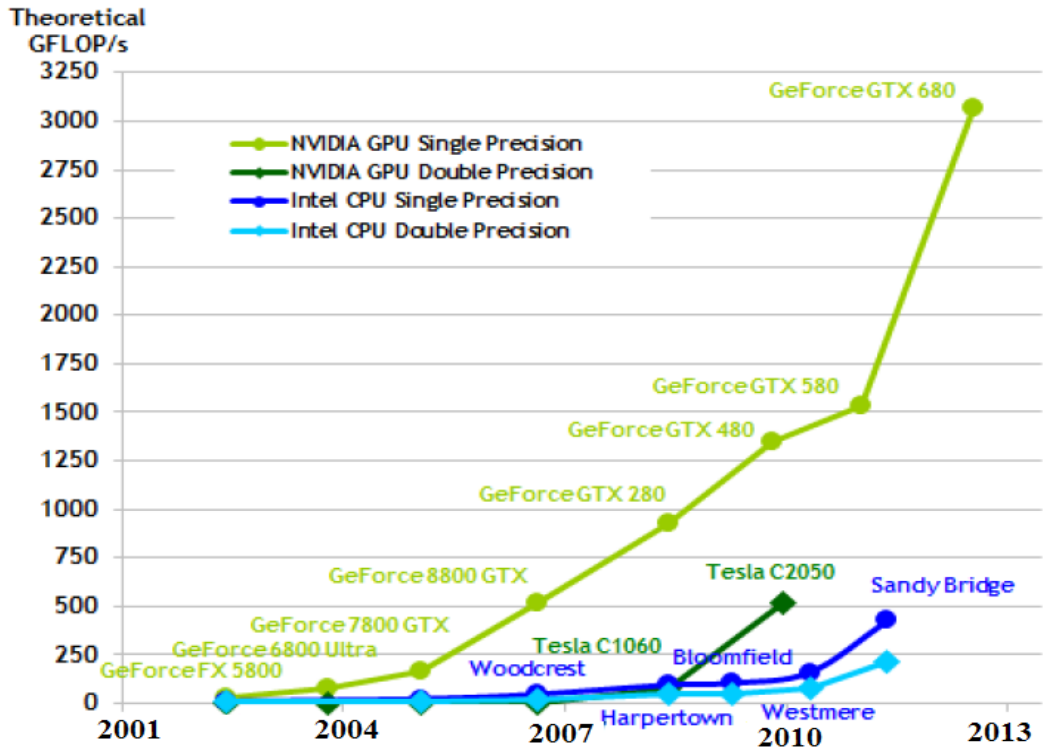

Figure 2.8: Floating-Point Operations per Second CPU vs. GPU [12]

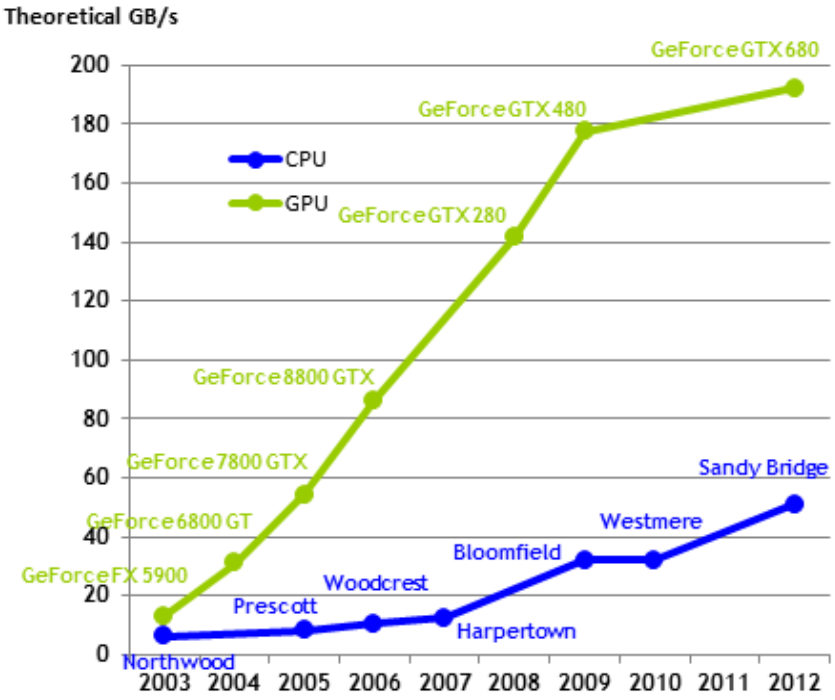

Figure 2.9: Memory Bandwidth for the CPU vs. GPU compared [12]

The computational difference, Giga floating-point operations per second (GFLOPs), between the CPU and the GPU is improving through time with an enormous rate. Apart from its use in graphics rendering applications in HD 3D applications, where enormous amount of pixels are mapped in to parallel cores of the GPU for rendering the above advantage has made GPU to attract the interests for post rendering multimedia applications like video compression, video processing etc.

2.2. LITERATURE REVIEW

Various researches have been done to improve the performance of automatic recognition, in terms of algorithm improvements and hardware software co-processing. This section summarizes the various works that have been done in the area of target recognition.

Algorithmic Improvement

Algorithmic improvements in automatic target recognition have been done through improving of one or more of the algorithms in the pipeline, for the overall improvement of one of the metrics of ATR.

Segmentation

The requirement for better quality segmentation, which by itself requires complex and computational intensive algorithms, with the time constraint for real-time applications has created a tightly constrained problem in segmentation. Numbers of researches have been done to address this problem from different perspectives:

One popular method that is developed in this context is the watershed algorithm [19]. This algorithm approaches the problem by converting lines in an image into “mountains” and uniform regions into “valleys” that can be used to help segment objects. Watershed algorithms first task is to take the gradient of the intensity image frame; which in effect forms valleys or basins (the low points) where there is a continuously homogeneous region and forms mountains or ranges (high points matching to edges of regions) when the algorithm encounters dominant contours in the image.

Then successive morphological flooding of the valleys is performed starting from application specific point on the image frame until algorithm specific requirements are met. Dispersed regions from the previous operation fill up during the successive morphological flooding operations. The merged regions across the marks generated above are segmented to belong to the same region as the image frame is filled by flooding operation. Following this approach, the valleys that are connected at that particular marker point become components of that marker. Then the image is segmented into the corresponding marked regions [19].

Watershed algorithm is flexible in that it allows an implementer of the algorithm (an algorithm developed based on watershed) to choose marker points, at a pre-specified region in the background, to be parts of the target or parts of the background that are known part of the target to be detected or part of background scene respectively. Doing so, the programmer is able to tell the algorithm to group certain sets of points together as targets or background or vice versa. The watershed algorithm generally segments the

image by allowing marked regions to incorporate the edge defined valleys in the gradient image that are then connected to segments of the region [19].

Another popular work on image segmentation was the Grab Cut algorithm [24]. Grabcut unlike mathematical morphology based algorithms [19][41] has a high accuracy but is computationally an expensive algorithm. The GrabCut creates the Background/foreground segmentation grouping the pixels into clusters of similar colors and introducing boundaries between foreground and background pixels. An Iterative Graph Cut algorithm is used to optimize process that tries to connect pixels with similar labels. This imposes a penalty for placing a boundary in regions of relatively uniform intensity, which is the source of the performance penalty of the whole algorithm.

A powerful image segmentation algorithm, Mean Shift Segmentation, was introduced in [41]. Mean shift is a nonparametric estimator of density which has been applied to image segmentation. It starts at each data point (aka pixel in an image or video) and first estimates the local density of similar pixels (i.e., the density of neighboring pixels with similar color).Using the density the algorithm estimates the local density gradient. These estimated density gradients are used within an iterative algorithm to find the peaks in the local density. Each pixel that is drawn upwards to the similar peak is considered to be members of the same segment.

Mean shift algorithm has significant limitation if the local characteristics of the pixels vary considerably across the domain; it is difficult to select common optimal influence region [41]. As a result of this, in a segmented image with varying color, some objects may appear too coarse while others are too fine.

Hardware software co-processing

The performance in the improvement of ATR attained using hardware co-processing is dealt with terms of different metrics of computation: Speed [38][37][34][39][40], Accuracy of result[36][37][40], power efficiency[35].

In [35], an FPGA based face recognition system which is reasonably power efficient than floating point architecture and can be employed for portable applications was designed using Eigen values. Fixed point implementation of the algorithm [35] gave an improvement in the power, by saving thousands of machine cycles at the cost of losing 0.008% weight in highest three Eigen values. The algorithm is a modified in such a way that the number of iteration required by the algorithm is significantly with a tolerable decrement in the precision of the algorithm. It shows a significant amount of reduction in power dissipation at the cost of a slight decrement in the accuracy of the algorithm.

In [36] a Bessel K forms target recognition method using Haar-based detector to select potential regions of interest within images was implemented. Detection and recognition limits are constrained under non-ideal environmental as well as sensor based uncertainties. A region based approach to identify targets using Bessel K form-based encoding was introduced. The same approach has been followed for performing clutter rejection to minimize the number of computation performed.

Inherent performance advantages of the Grayscale Optical Correlator(GOC) is taken advantage of in [37] using Neural Networks based ATR application. The performance advantage from GOC is used to decrease the false positive detection rate while retaining the high positive detection rate obtained by the GOC. In their currently ongoing project

they are designing new GOC hardware system architecture which relies on the utilization of their maturing Digital Light Processor (DLP) as both the input and the filter Spatial Light Modulator (SLM). They are also working advancements in the ATR processing algorithm for the new architecture.

In [38] implementation of a template matching algorithm on Synthetic Aperture Radar (SAR) was developed at Myricom for performance improvement. The system was migrated to the multiple, parallel, FPGA computing nodes connected by Myrinet. These FPGA nodes achieve high efficiency, through the exploitation of the systolic nature of the ATR algorithm on the FPGA devices.

Performance improvements also for the morphological operations in the ATR algorithm have been done in some works [34]. A two level compilation scheme was used for producing high speed binary image morphology. A pre-specified set of instructions is generated by the first compiler which is primarily suited for the second compilers instruction set. Since the synthesis on FPGA operation, is replaced by the two levels compile operations, a ten times speed up was achieved for the morphological operation [34].

Another FPGA based partly scalable design of ATR system was designed in [39]. A linear systolic implementation of the ATR algorithm is mapped to the Splash 2 FPGA platform. The column oriented processors with distributed Splash 2 memories were used throughout the design to achieve the high performance and the scalability of the system.

The NetBots algorithm is used to improve accuracy in ATR by introducing a social network to the swarm system [40]. A general accuracy improvement was gained using

the social network keeps track of the target detection histories and is used to estimate the accuracy. Implemented on robotic systems, the accuracy of the detection is determined by the previous accuracy results to give weight for each of the robots detection and agree with the majority.

Since the emergence of general purpose GPU, the computer vision industry has transferred its attention to the processing power of GPU devices to fulfill the ever increasing computational requirement of computer vision applications.

In [40] a real time computer vision based power line extraction solution is considered for active Unmanned Aerial Vehicle (UAV) guidance. A collinear line segments fitting algorithm considering global and local collected data together with multiple collinear measurements. NVIDIA GPU is used to speed up the implementation of the algorithm to outperform previous baseline line detection algorithms, for real time application.

CHAPTER 3

SYSTEM DESIGN AND IMPLEMENTATION

In this chapter the methodology followed for the implementation of the thesis is discussed. The main part of the algorithm actually starts after an RGB image sequence is obtained from an appropriate video source, video file or digital camera.

The algorithm for this work incorporates: Capturing Frame, Object Detection/Segmentation, Target Recognition and Target Tracking. Capturing frame involves accepting input video from camera or input video file. The image segmentation phase minimizes the search area for the target recognition phase by getting the region of interest on the captured frame. The target recognition phase then recognizes the target by comparing the ROI's detected from the segmentation phase with predefined target templates.

The target template used is a template which can be used as an identifying feature of the particular target. The target template, which basically can be either a shape based identifying feature of the target or color based identifying feature of the target, is compared with the potential targets using template matching algorithm. The target template used for this work is a composite template which is a combination of the color feature of the target and the shape feature of the target.

3.1. Target Detection

This section focuses on how to isolate objects or parts of objects from the rest of the image. For efficient resource utilization the region of interest has to be detected before executing further otherwise unnecessary processing is done.

In video security, for example, the camera mostly looks out on the same static background, which really isn't of interest for the surveillance application. What is of interest is when target enter the scene, or even when something which was part of the background is missing. Rather than doing target recognition to the whole area of the extracted frame from video input, it is efficient to do template matching to this areas of interests. So, in order to achieve this first we have to isolate the events before doing the actual target recognition stage.

For this work the target detection is a composite process which includes color segmentation and background subtraction; which separately are weaker algorithms[19][24][41], but when combined[1] give us a better performance.

3.1.1. Color Segmentation

In target recognition apart from isolating objects from the rest of an extracted frame we want particular objects to be recognized and tracked. In this work template matching is performed for target recognition. One part of the target template is color (Hue Part of an HSV image); this information is used for the segmentation/detection phase in order to minimize the search area of the target identification/recognition phase.

The extracted frame image has to be converted to HSV image space before color segmentation is done to determine whether each pixel resides in the range specified in template or not.

RGB to HSV color space

The HSV system is based on a warped version of the RGB space and is directly related to intuitive color notions of hue, saturation and intensity. The conversion from the RGB color space to the HSV is done using the following equations:

$$h' = \begin{cases} \frac{g-b}{\delta} \; if \; r = max \\ \frac{2+(b-r)}{\delta} \; if \; g = max \\ \frac{4+(r-g)}{\delta} \; if \; b = max \end{cases} \tag{3.1}$$

$$v = \max \tag{3.2}$$

$$s = \frac{max - min}{max} \tag{3.3}$$

$$h = 0.167 * h' \tag{3.4}$$

$$where: max = max(r, g, b)$$

$$min = min(r, g, b)$$

$$\delta = max - min$$

For $r, g, b \in [0 \ldots 1]$, Equation 3.1– Equation 3.4 gives the corresponding result $h, s, v \in [0 \ldots 1]$. The hue parameter of the target template contains the minimum and maximum hue values ($< Minimum\_Hue, Maximum\_Hue >$ pair) contained in the target. This range could be small or large depending on the color content of the target. Usually targets

are composed of a single color; for those situations the Hue range will be very narrow. The $Range\ (Frame)$ operation will give a bitmap, which has for each pixel:

$$Hue_{Frame} = Range\ (Frame) \tag{3.5}$$

$$Hue_{Frame(x,y)} = \begin{cases} 1, & if\ Minimum_{Hue} < Frame(x,y) < Maximum_{Hue} \\ 0, & Otherwise \end{cases} \tag{3.6}$$

The result of the Range operation is used as a mask to threshold the result of Background subtraction Phase.

The discontinuity in the Hue component of the HSV image on Red (around 0 and 360°)which is at the division where Hue 'rolls over', is dealt with by always setting hue values to be within the 0-360° range using modulus arithmetic operation[42]. So, whenever the hue range being considered contains the discontinuity, the above operation is modified as:

$$Hue\_Frame(x,y) = \begin{cases} 1, & if\ Minimum_{Hue} < Frame(x,y) < 360 \\ & \text{or } 0 < Frame(x,y) < Maximum\_Hue \\ 0, & Otherwise \end{cases} \tag{3.7}$$

3.1.2. Back Ground Subtraction

Background subtraction is a popular motion detection technique [1]. It operates by simple image subtraction of the current frame from the background to find the foreground moving objects. This technique is very efficient for static camera configuration. But for moving camera, the background changes, this technique has to be assisted by other techniques to compensate the movement such as dynamic morphological operations.

The term “background” can be defined in different contexts depending on its applications. For instance, for a highway application the average traffic flow is considered background [2]. Generally, background can be defined as any static or periodically moving parts of a scene that remain static or periodic over the period of interest. For this reason background modeling should be done before background subtraction.

Most background modeling algorithms have the drawback of, a weak assumption that is often violated: all pixels in a frame are independent. However, the methods to consider neighboring pixels come at the cost of twofold [2] of extra memory and computation. For this reasons these methods are not practical.

In this work, Averaging background method is used in conjunction with noise cleanup techniques: Median Filter for smoothening, cascaded Erosion and Dilation for noise cancellation and Flood Fill for filling holes in the segmentation.

Averaging Background

Averaging background method involves learning the average and standard deviation of each pixel to model the background. For this purpose the operations $Accumulate()$ to accumulate learned background, which gives an approximate average of the background. $Absolute\_difference()$ is used to find the absolute value of the difference of the current frame and the average background. And Threshold() operation is used to segment in to background and foreground models taking the difference more than a pre-specified threshold value as part of the segmented value and below that as part of the background.

The mean is approximated by a running average method which is given by:

$$Accumulate(x,y) = (1 - LR) * Accumulate(x,y) + LR * Image(x,y) \quad (3.8)$$

Where, LR is the learning rate of the algorithm

The current foreground object is found by doing an absolute value difference between current frame and the average background, which is found by the $accumulate(x,y)$ function.

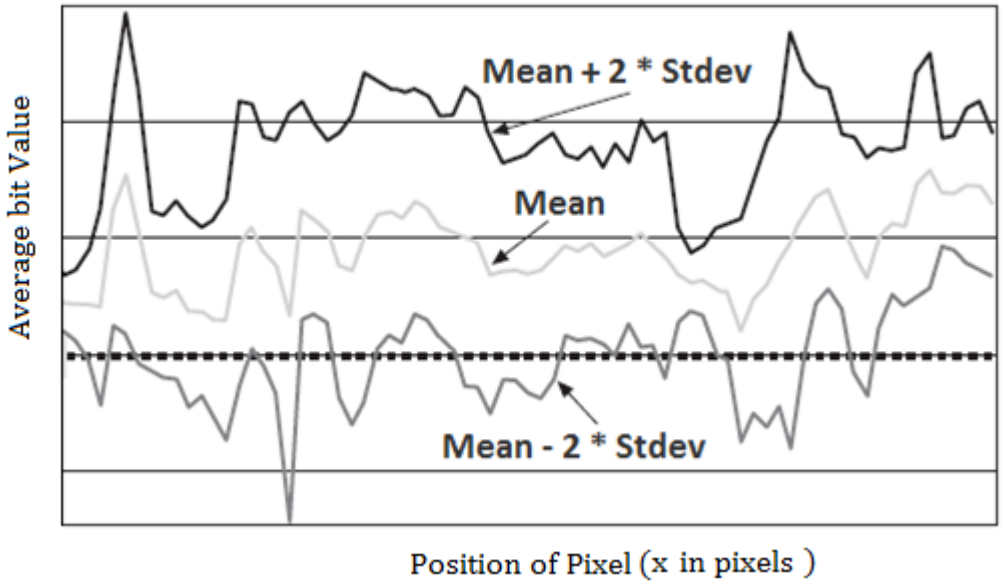

Figure 3.1: Averaging method with High/Low Thresholds.

The standard deviation can be formulated as:

$$\sigma^2 = \frac{1}{N^2}\sum_{i=0}^{N-1}(x_i - Mean(x))^2 \quad (3.9)$$

The problem with this equation is that it needs making one pass through the images to compute $Mean(x)$ and then a second pass to compute σ^2. However, after simple algebraic manipulations, it can be approximated by:

$$\sigma^2 = \left(\frac{1}{N}\sum_{i=0}^{N-1} {x_i}^2\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1} x_i\right)^2 \quad (3.10)$$

Using this form, we can accumulate both the pixel values and their squares in a single pass. Then, the variance of a single pixel is found using the average of the square minus the square of the average.

Morphological Operations

The difference image found using the above operation contains any difference found between the current frame and the modeled background, which contains salt and paper noises due to camera uncertainties, background movements, and etc. The camera movement will result in slight difference at the edges of the objects, because of these the difference to avoid that it passes through noise cleaning, this involves Passing through Median Filter.

$Median\_Filter(in\_frame, out\_frame, val)$:

$$Out\_frame(x,y) = Median(in\_frame(i,j)), for: \begin{cases} x-(val-1)/2 < i < x-(val+1)/2 \\ y-(val-1)/2 < j < y-(val+1)/2 \end{cases} \quad (3.11)$$

This is followed by DEED (Dilate, Erode, Erode, and Dilate), since the median filter operation could not avoid the salt and paper noise [11].

Next phase this image is passed through a thresholding function which labels any difference big enough as foreground object:

$$Threshold\ (extracted\_frame, foreground\_frame, th)$$

Where th is selected based on experimental resultto give a reasonable detection rate with a reduced false-detection.

$$foreground_{frame(x,y)} = \begin{cases} 0\ if\ extracted_{frame} \geq th \\ 1\ if\ extracted_{frame} < th \end{cases} \quad (3.12)$$

The result of the Threshold operation is merged with the result of Hue_Range() operation which gives the final segmented frame, which has all the possible targets. In this stage the bitmap result from Hue_Range() operation is used as a mask for thresolding the foreground frame:

$$Result_{(x,y)} = \begin{cases} foreground_{frame(x,y)}, if Hue_{Frame(x,y)} = 1 \\ 0\ , if Hue_{Frame(x,y)} = 0 \end{cases} \quad (3.13)$$

3.1.3. Extracting possible targets from the segmented image

The result of target detection phase is a binary map. The binary map is then passed through morphological filters. The result of the two consecutive operations is basically a bitmap which for every pixel has a value of 1 or 0 depending on the particular position of the pixel contains a detected target or not. 1 represents the pixel points that belong to the target detection positive result points and 0 represents that at that particular point there is no target detected. The next task performed is transforming the bitmap result so found to regions, connected pixels with the same value (1) aka blobs, which are the outputs of the target detection phase.

To do the above task two different approaches are followed by the two popular Vision Frameworks: OpenCV and CVBlob. OpenCV offers $FindContours$ and $CVBlobs$ offer $BlobExtraction$ for this job**.**

Since target extraction job is a time consuming job, unless done carefully, a performance comparison is made to choose between the two libraries to use for this job. The performance comparison was done based on the data obtained from [30].

The comparison was done on a 64 Bit desktop computer with Core i3-2100 CPU @ 3.10GHz and a RAM 4 GB on a Windows 7 operating system. The OpenCV 2.4.2 library obtained from [30] was built by CMAKE [32], an open-source build system. The CVblob library cvblobslib_OpenCV_v8_3, which also was built from the source found at [31] using CMAKE.

The dataset obtained from [30] included different sized images with different amount of varying sized blobs. The images taken for this test are square images and the blobs are circular blobs and similarly spaced. Figure 3.2 shows an example test image taken for an image containing 36 blobs:

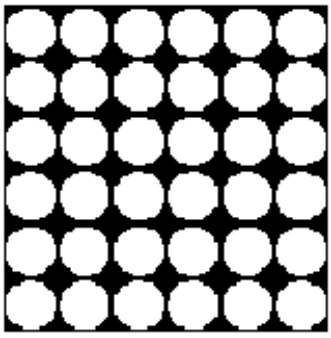

Figure 3.2: Example Image taken for performance Comparison of target extraction algorithms.

Images of varying sizes: 160 x 160 pixels

$$320\ x\ 320\ pixels$$

$$480\ x\ 480\ pixels$$

$$640\ x\ 640\ pixels$$

$$1280\ x\ 1280\ pixels$$

Containing 3×3(9), 4×4(16), 6×6(36), 8×8(64), 10×10(100), 12×12(144), 14×14(196) circular blobs. A total of $4 \times 7 = 28$ square images were used for the performance comparison.

Blob analysis result (ms)						
Number of Blobs	Image size (Px. X Px.)					
		160 x 160	320 x 320	480 x 480	640 x 640	1280 x 1280
	9	6.33	8.23	13.24	22.9	95.33
	16	6.87	8.67	13.33	23	95
	36	6.99	8.87	13.33	23.87	95.8
	64	7.33	9	14.22	24.2	96.33
	144	7.33	9.22	14.33	26.33	97
cv::findcontours analysis result (ms)						
Number of Blobs	Image size (Px. X Px.)					
		160 x 160	320 x 320	480 x 480	640 x 640	1280 x 1280
	9	7.09	9	15.22	25.14	99.67
	16	8.24	10.33	15.6	26.22	107.33
	36	8.59	10.73	15.73	27.67	110.17
	64	9.24	11.07	17.182	28.55	113.67
	144	9.53	11.52	17.48	31.6	116.33

Table 3.1: Performance of cvblob (top) and performance of findcontours(bottom)

A program for the comparison was developed for testing of the performance of each of the algorithms independently. Both the algorithms were tested on each of the blob detection algorithms three times each. The mean of the three computation times was taken.

From the performance comparison result found, it is conclusive that cvBlob outperforms $findcontours$ algorithm, Table 3.1. The difference in performance between the two algorithms increased as the image size increases. For a given image size increasing the number of blobs also increased the difference in performance between the two algorithms. It is also worth noting that the ratio in performance of the two algorithms decreases as the image size increases; the highest performance ratio being for the smallest frame size containing maximum number of blobs, 1.31.For this work the cvBlob library frame work is used for blob detection, for the aforementioned performance achievement over OpenCV's $findcontours$ algorithm.

From the array of blobs extracted from the above operation, we get the rectangle bounding the potential target objects and that would be the region of interest (ROI) to our recognition phase, but it could contain very large and very small to be a target. These frames were eliminated by comparing with the largest, 1000, and smallest, 50, possible blob sizes and taking what is in the range only, *eliminateLSconoturs(countourArray).* The result of this operation is possible targets only, which is an input to template matching.

3.2. Template matching

Templates of a target contain templates of the targets at varying sizes and directions. Size of particular extracted frame (ROI) is used to select particular templates to correlate with the detected potential targets.

For this work, target templates are produced in such a way that to contain 6 pixels extra in every direction (Top/Bottom/Left/Right) of the template. So, the output contains 13 x 13 pixel normalized correlation output Matrix whose values range from 0(total mismatch) to 1(Full match). In this work a match greater than 0.5 is accepted, because of the assumption taken object match value more than half is considered to recognize targets which are partially, less than 50% , obstructed. The output of the Recognition (template matching) phase is the maximum correlation result from the correlation output matrix; Depending on whose value target is either recognized or not.

This part of the system has high computational requirement, and high degree of data independency. This makes the template matching phase prone to parallel processing through Co-processing with devices with high parallel processing capabilities to get a very good speedup. GPU is the best candidate, as mentioned in section 2.1.3, due to the nature of the problem.

Algorithm3.1 shows the template matching algorithm used for this work, given an input Image Frame and Target Template gives the best matching position and the normalized match quality.

MatchTemplate

$Input: Frame, Template$

$Output: Match_{Qualty}, Match_{Position}$

$for\ i\ in\ 0\ to\ Frame.Height - Template.Height + 1$

do

$for\ j\ in\ 0\ to\ Frame.Width - Template.Width + 1$

do

$Quality \leftarrow Compute\big(Frame, Template, Position(i, j)\big)$

$if\ Quality > Mathc_{Quality}$

$then$

$Match_{Position} \leftarrow Position$

$Match_{Qualty} \leftarrow Quality$

$End\ if$

$End\ for$

$End\ for$

$Return\ Match_{Position}, Match_{Qualty}$

End ***MatchTemplate***

Algorithm 3.1*: CPU based implementation template matching algorithm*

$\boldsymbol{Compute}(Template, Frame, Position)$

$Output \leftarrow Quality$

$Intermediate\ Variables\ temp1, temp2, temp3 \leftarrow 0$

$for\ x\ in\ 0\ to\ Template.Heght$

do

$for\ y\ in\ 0\ to\ Template.Width$

do

$temp1 \leftarrow temp1 +$

$[Template(x, y) - Frame(Position.i + x, Position.j + y)]^2$

$temp2 \leftarrow temp2 + [Template(x, y)]^2$

$temp3 \leftarrow temp3 + [Frame(Position.i + x, Position.j + y)]^2$

$End\ for$

$End\ for$

$Normalization_{Factor} \leftarrow sqrt(temp2.temp3)$

$Quality \leftarrow temp1/Normalization_{Factor}$

$Return\ Quality$

$End\ \boldsymbol{Compute}$

Algorithm 3.2: Computation for template matching algorithm

3.4. GPU Implementation

Considering GPUs are intended to enhance the performance of particularly data parallel applications, such as the ones in graphics, the suitable applications for GPUs are the ones that feature data independence between parallel computational units. Most of vision algorithms, like Template Matching, match this requirement. The performance bottleneck for most GPU based data parallel algorithms is memory access, including initialization and write back operations. NVIDIA's GeForce 8400 GS architecture also presents a performance sensitive memory hierarchy, Figure 3.4. Algorithm implementation has to be carefully designed to optimize memory performance

CUDA Vs. OpenCL.

CUDA and OpenCL are the two most popular GPU programming frame work s that are used to program GPUs for general purpose computation, particularly computation with high degree of parallelism.

OpenCL is an open standard that runs on the device and is developed by multiple vendors AMD, ATI and NVIDIA. In OpenCL for every device running different kernel execution is required. And for every kernel execution it uses its own C-like language, while it uses pure C for its kernel management.

NVIDIA's CUDA offers a straightforward framework to handle data-parallel computations. It is more abstract and high level than OpenCL. It is much more mature and has more convenient high level Application Program Interfaces (APIs). No special

code is required for thread/process management as these are done entirely by the GPU. CUDA also offers very flexible control over memory access. Most importantly CUDA executes much faster on NVDIA GPUs than OpenCL. Because of the aforementioned reasons, we chose CUDA to be the right frame work to implement and analyze the template matching algorithms on GPUs for this work.

CUDA programming model

CUDA Toolkit is intended to applications whose control parts is executed on general purpose processors(Host), and one or more data intensive parts is executed on NVIDIA GPUs as coprocessors for accelerating Single Instruction Multi Data (SIMD) parallel jobs. This process is known to be self- contained [22]. It means data intensive parts are executed by a batch of GPU threads entirely without intervention by the host processor, taking advantage of the parallelism from the parallel graphics hardware. Executions performed on the GPU jobs are considered as remote procedure calls by the host processor using the CUDA Toolkit.

CUDA Background

The CUDA architecture is built on an array of multithreaded Streaming Multi-processors (SMs), which are each designed to execute hundreds of simultaneous threads. When a program hosted CPU executes a CUDA ”kernel”, which are sections of computation that are executes in parallel on the GPU, threads are instantiated and spread on the SMs with available execution capacity. Threads are arranged in a hierarchy of blocks and grids as shown in Figure 3.3 below.

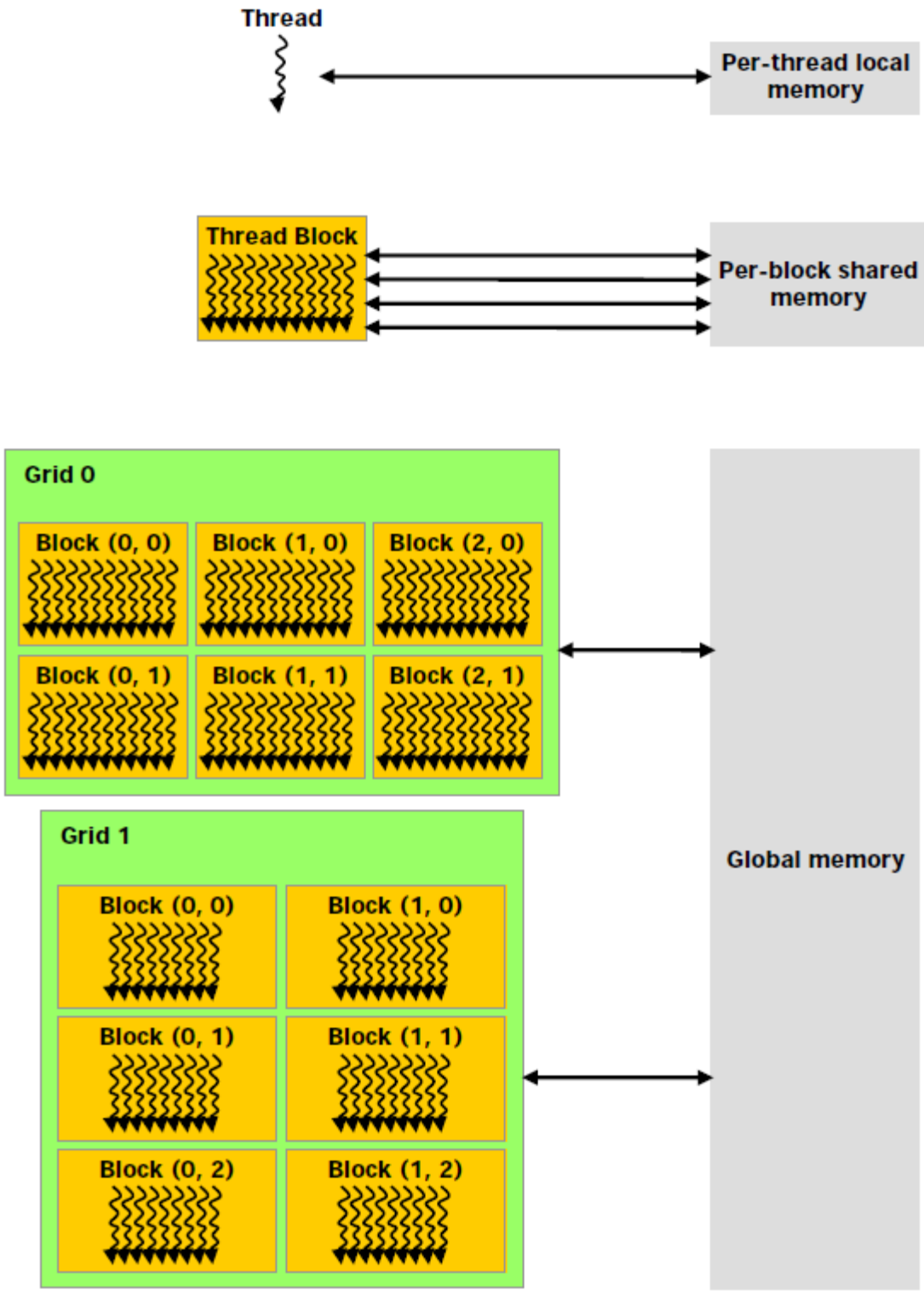

Figure 3.3: CUDA programming model of threads [13]

Each thread executes an instance of a kernel function and has a unique thread number and execution resources, including program counter (PC) besides registers and dedicated local memory [13]. A block is a set of threads running in parallel that have shared memory amongst them and can synchronize within certain barrier constraints. Each block has a unique block number that provides as an identifier within a grid, which is a block of blocks [13]. Grids share global device memory and perform thread management and synchronization between successive kernel calls.

The threads in one block execute in parallel on a single SM, and multiple blocks can execute on a single multiprocessor. As the execution of blocks terminate, new blocks are launched on the multiprocessors that finished execution [13]. The abstraction of this operations offered by CUDA, allows to do automatically scalable CUDA code as CUDA typically allocates threads based on problem size and the available resource implicitly. The computation can be done without considering target specific hardware implementation constraints. Given a particular CUDA based implementation devices with more capability, with a greater number of SMs, are able to process more of the threads concurrently, leading to faster execution times with minor target specific code revisions.

Figure 3.4 shows CUDA hardware model architecture with processors, registers and hierarchical cache on Multiprocessors. NVIDIA calls this architecture SIMT (Single Instruction, Multi-Thread), which is similar, and usually implementation, to the popular SIMD. They share the characteristics that a single instruction operates on multiple processing units.

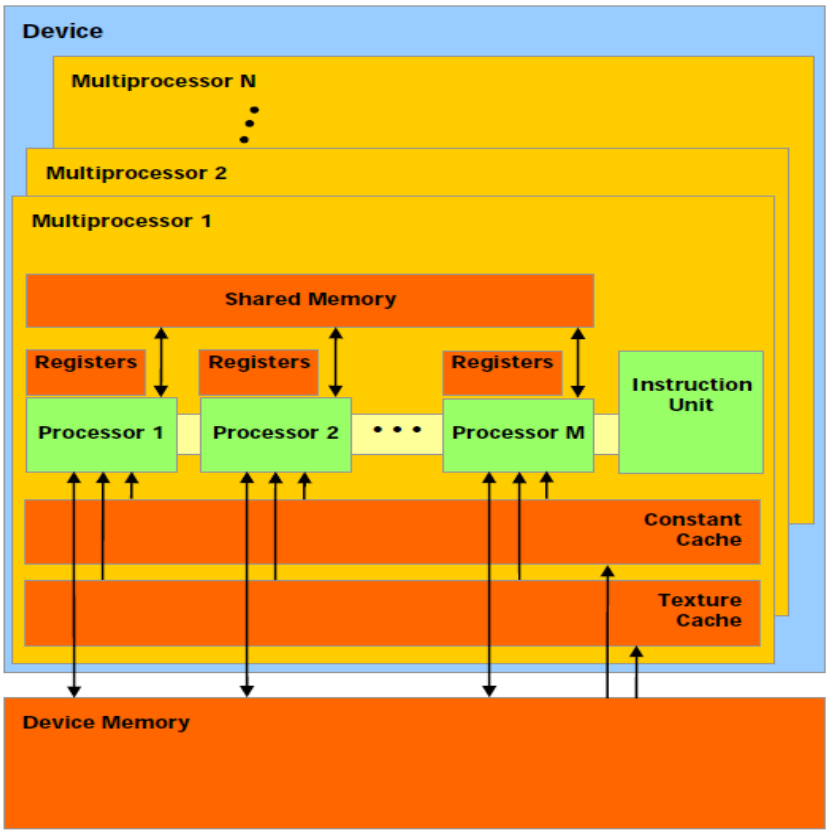

Figure 3.4: CUDA hardware model NVIDIA GeForce 8X [13]

Besides their analogy, the essential variation between the two is that SIMD operations render the multi-data to the software, while SIMT instructions explicitly indicate the execution and control flow behavior of a single thread [13]. SIMT facilitates computations to be coded allowing thread level parallelism for single and autonomous threads, apart from data parallel computations with coordinated threads.

GPU Implementation

The GPU implementation distributes the operations to be done on different threads to be executed on the device. Each thread is executed in different GPU cores in parallel to speed up the process. Algorithm 3.3 shows the GPU implementation of the template

matching algorithm. The computation operation is the same as the computation operation for CPU, Algorithm 3.2, and the only difference in this case the computation is performed on the device.

$\boldsymbol{ComputeGPU}(Template, Frame, Position)$

$Input: Frame, Template$

$Output: Match_{Qualty}, Match_{Position}$

$i \leftarrow ThreadID$

$x = i \,/\, (Frame.Width - Template.Width + 1)$

$y = i \,\%\, (Frame.Width - Template.Width + 1)$

$Quality \leftarrow Compute\big(Frame, Template, Position(x, y)\big)$

$if\ Quality > Match_{Quality}$

$then$

$Match_{Position} \leftarrow Position$

$Match_{Qualty} \leftarrow Quality$

$End\ if if\ ThreadID = 0$

$Normalization_{Factor} \leftarrow sqrt(temp2.temp3)$

$Quality \leftarrow temp1/Normalization_{Factor}$

$Return\ Quality$

$End\ \boldsymbol{ComputeGPU}$

Algorithm 3.3: GPU based implementation template matching algorithm

Figure 3.5 shows the high level flow chart of the overall ATR system. It includes two major phases, detection and recognition. In the detection phase, two separate segmentation algorithms are done, color based segmentation and background subtraction based segmentation. In the recognition phase, the detected Blobs are matched against, the target templates, which could be done either in CPU or GPU. After the recognition phase the post recognition operations, marking the target and tracking, are performed.

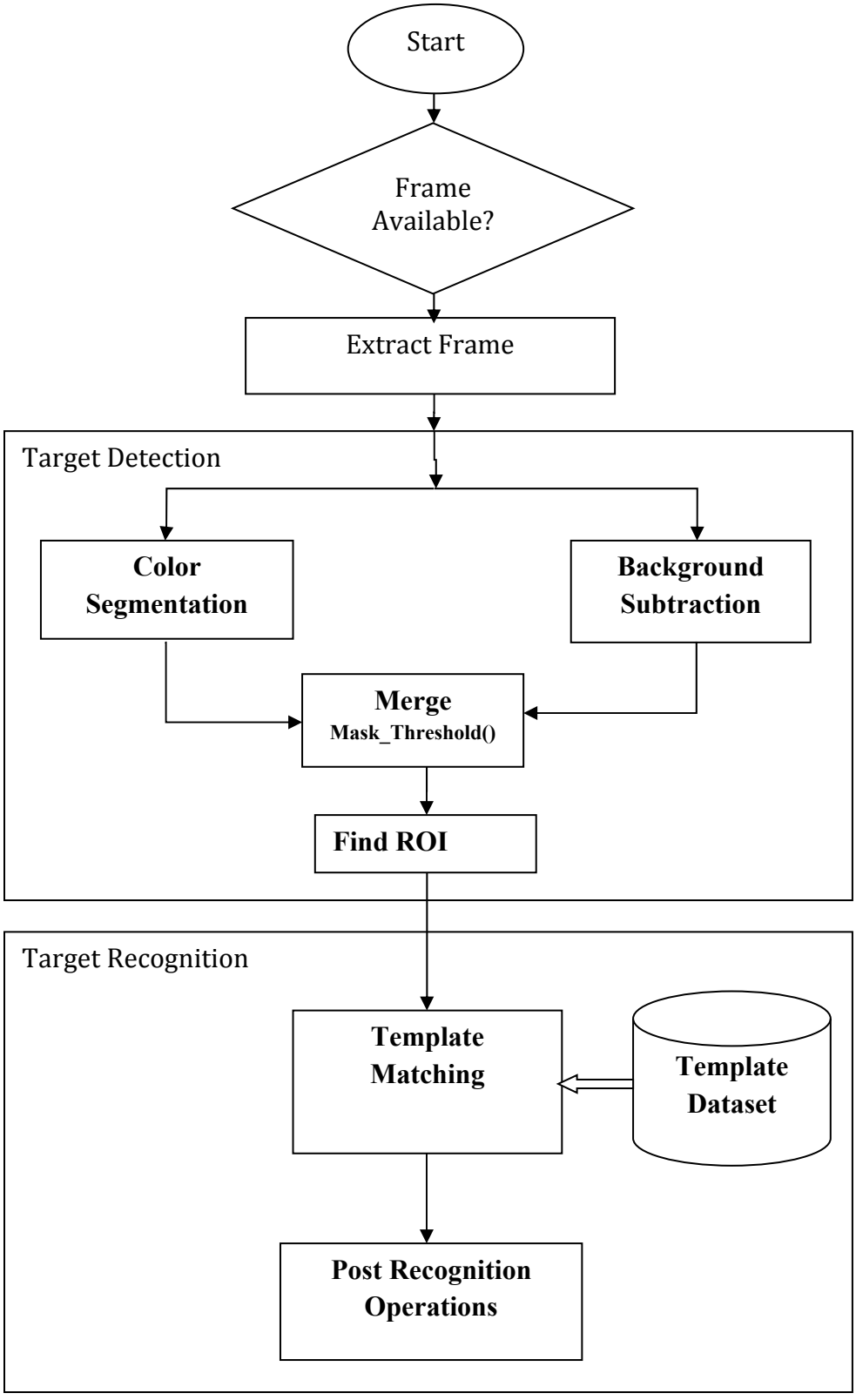

Figure 3.5: High Level Flow Chart of the ATR system

CHAPTER 4

Experiments and Results

4.1. Experimental Setup

The algorithm is tested on an Intel Core i3-2100 CPU at 3.1 GHz frequency, with CUDA enabled GeForce 8400GS GPU. The GPU has 8 CUDA cores, with a processor clock of 1340 MHz, 64 bit memory interface for a dedicated video memory of 1024 MB and PCI Express x16 Gen2 bus.

Test Data

The algorithm is tested on four video sets, Ideal Target, Simple Background, Complex Background and Similar Colored Background; for simple Targets and Complex targets. Two types of lighting conditions were considered, Indoor and Outdoor. The test set included video files, with frame rate of 25 FPS and have a resolution of VGA (640 x 480 pixels) for all experiments, except for the speed up test which used resolutions in table 4.4. The test scenarios used are explained as follows:

- Ideal: This is a flash generated video, unlike the rest of the test sets, file which is free of any camera generated noises and the target is differently colored with respect to the background.

- Simple Background: This is a video file which has a plain background; it is considered both in Indoor and Outdoor scenarios.

- Complex Background: This is a video file whose background contains moving and stationary non-target objects. This is also considered in both Indoor and Outdoor lighting scenarios.
- Similar Colored Background: In this scenario the videos has background which has the same color as the target. This was also considered in both Indoor and Outdoor lighting scenarios.

The data sets considered for the experiment included different colored, different sized, different shaped in the selection. For checking robustness of the algorithm noisy targets are considered, since they have generally been one of the common problems in the area of automatic target recognition.

We then focused testing of the dataset on the target detection algorithm as implemented in Algorithm 3.1. Then the shape based target recognition algorithm, as implemented in Algorithm 3.2, for recognizing the targets in video frames was tested. The target templates are saved in the target dataset for the template matching process. This algorithm is triggered by a positive detection result from the target detection algorithm; the recognition process is carried on once the pre detected region is fed in to this algorithm.

4.2. Performance metrics used

In this section the performance metrics used in this thesis are described. Performance metrics used include metrics used for testing detection and recognition of the algorithm. Computation time: This metric represents the computation time required to process a single frame. The average of 3 tests is taken to consider performance fluctuations.

The metrics used for testing the detection phase are:

- Detection rate, which is the ratio of number of target pixels detected to the total number of target pixels available in a frame. The average is taken for the frames in the videos.
- Miss rate, is the ratio of missed/undetected target pixels to the total number of target pixels available in a frame.
- False detection is the ratio of number of non-target pixels detected to the total number of pixels available in a frame.

The metrics used to test the recognition phase are:

- Degree of recognition represents the result of the recognition phase, a normalized template matching algorithm result. It has a value between 0 and 1, 0 being total mismatch and1 full match. A degree of recognition of 0.5 is taken as a threshold for classifying recognized targets, to take partially covered targets in to consideration. If a non-target part of a frame is detected and recognized with a degree of recognition greater than or equal to 0.5, it is a false recognition.
- False recognition rate is the ratio of frames with false recognition to the total number of frames.
- Speedup is a metric used to compare the implementation of the algorithm on CPU and GPU. It is the ratio of the computation time of the implementation on CPU to the computation time of the implementation on GPU.

4.3. Results

The algorithm is tested for different performance metrics varying the important parameters of the algorithm. The detection phase is evaluated by varying the threshold value of the detection phase, to choose an acceptable level of threshold.

Table 4.1 shows the results found considering varying threshold values. A standard VGA resolution video is used for this experiment. The average of 3 consecutive time tests and the percentage of miss rate in the scene are considered. With an increase in the threshold value the speed of the algorithm deteriorates, while the number of target misses is minimized. As the threshold values increases the number of pixels that reside in the threshold range increases, by which increasing the number of further computations done, so does the time taken per frame. On the other hand for small threshold value the detection algorithm misses actual targets in the scene.

Threshold Value	Average Time/ Frame(ms)	% of Target misses
0	2.3	98
1	4.6	94
2	6.3	93
3	6.3	91
4	8.0	83
5	9.2	63
6	10.2	66
7	13.7	45
8	14.2	36
9	17.1	31
10	19.8	12
11	21.2	8
12	24.9	5.5
13	25.4	3
14	34.6	2.5
15	49.7	2
16	59.6	2
17	62.7	1.5

Table 4.1: Performance of the system for varying threshold values\

Since the threshold value being considered only affects the result of the background subtraction phase, it is scene independent. To make the algorithm in real time, average

time per frame of25 ms, within an acceptable miss rate, misdetection rate of3%, a threshold value round of 13 is chosen and used for the rest of the experiments used.

The detection phase of the algorithm, which contains both shape and color based algorithm, is tested for the effect of color range on detection rate and false detection rate.

In this test the performance of the algorithm, in terms of the false detection rate and detection rate, were measured for target templates of varying Color range, from template containing a single color to target templates containing end to end color combination in the Hue range. The test data taken for this experiment consideration with its respective histogram range representation is as shown in the Table 4.2below.

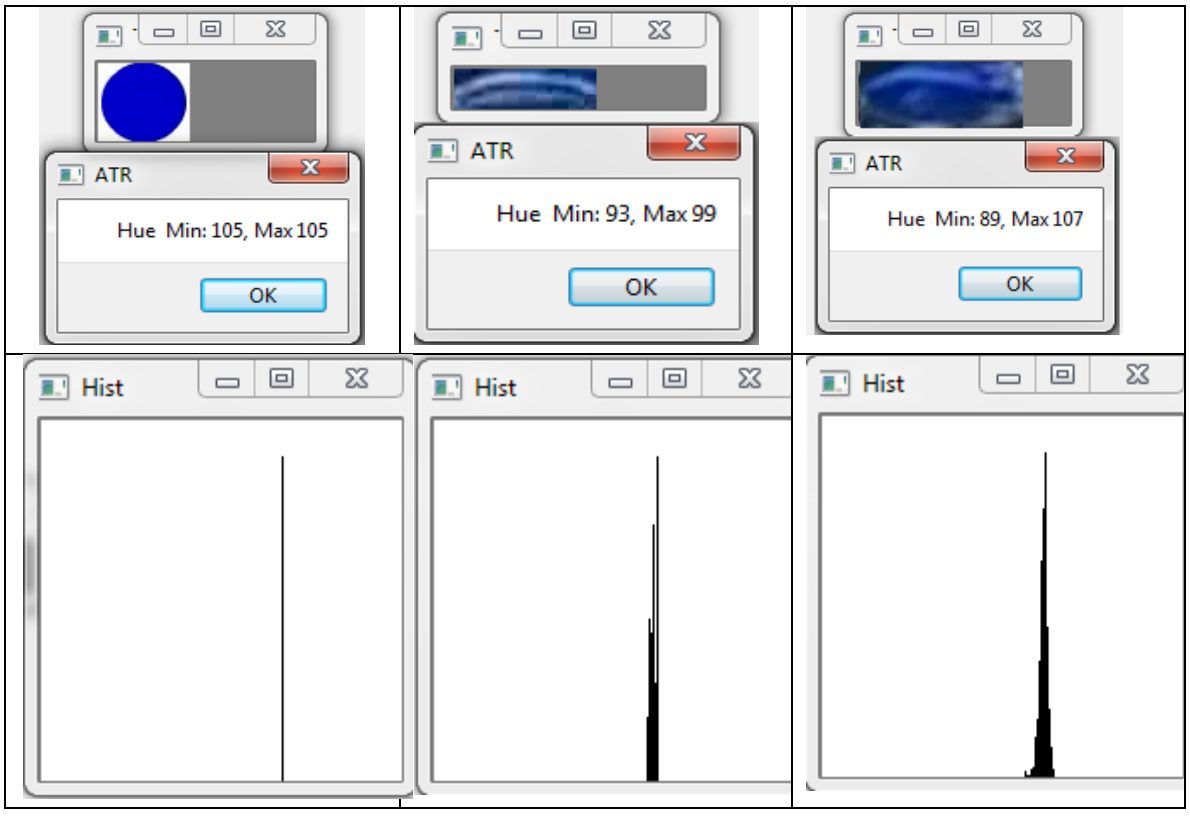

Table 4.2(1): Targets with varying color combination

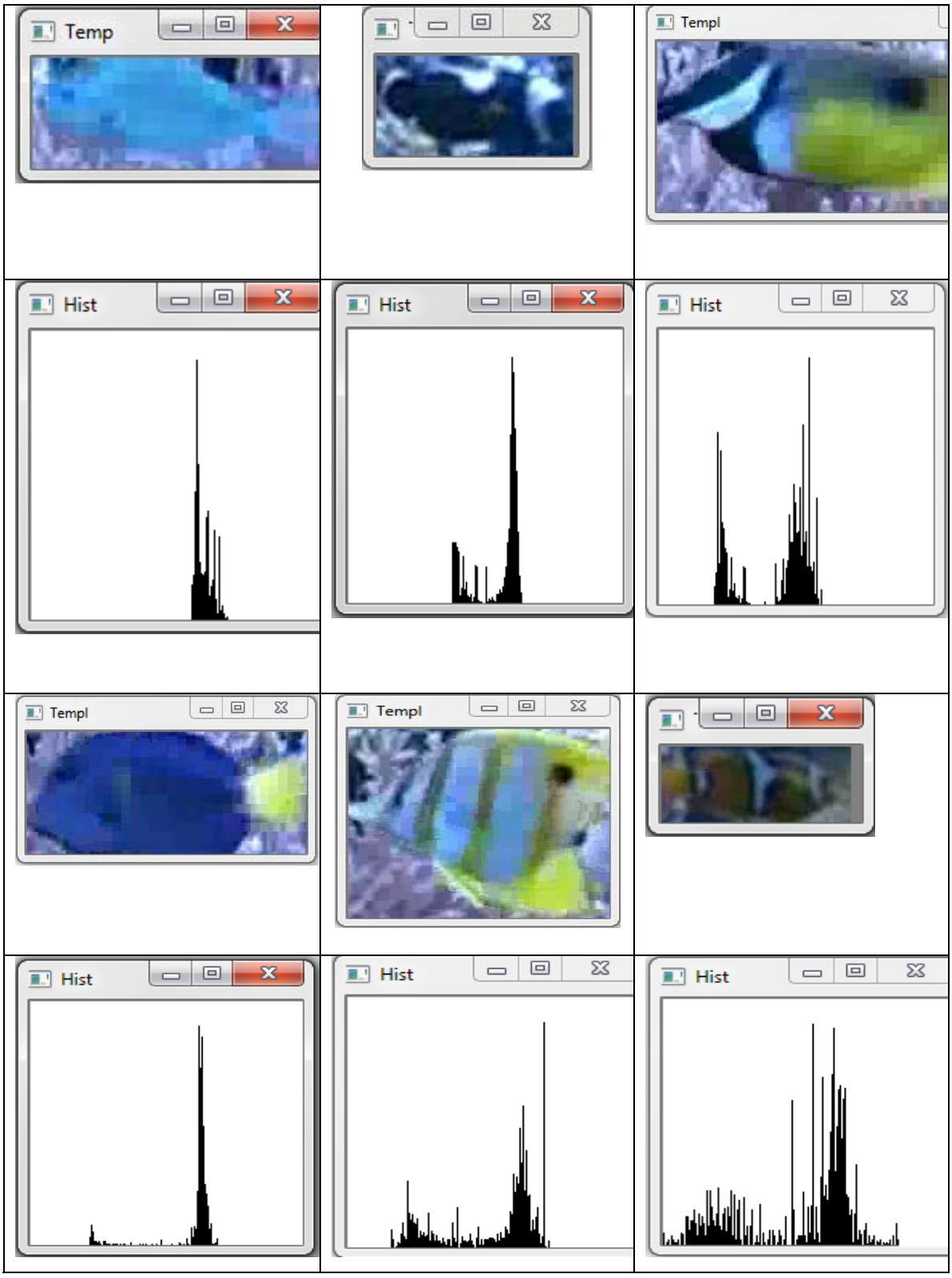

Table 4.2(2): Targets with varying color combination

The variation of the detection rate/false detection rate as a function of the color range is shown in the figure below, the result of the experiment on the data in Table 4.2. For data with narrow hue range the algorithm performs accurately, with no false detections and with all the target pixels detected. As the color range of the test data increases since more and more values get in the range the false detection rate increases and the detection rate considerably deteriorates, since the pixels at edges of targets are missed being parts of the non-target part of the background.

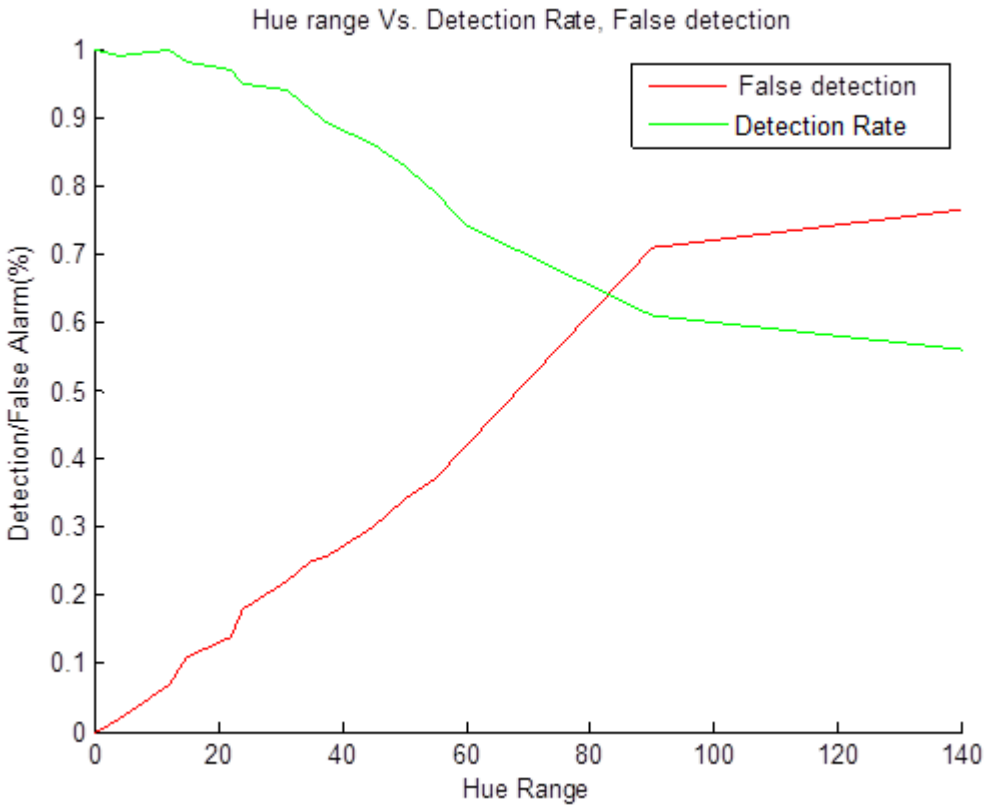

Figure 4.2: Variation of detection rate with Color range.

Depending on the color content of the target template the detection rate varied from 1 to 0.6 for hue ranges 0 and 140 respectively.

Next the target recognition phase of the algorithm is tested for varying scene and lighting conditions and varying sizes of targets templates.

Degree of Recognition for varying scenes

In this test the degree of recognition is considered for different scene types both for indoor and outdoor lighting conditions. The performance is measured in terms of the average time taken per frame, showing the time required for every frame being considered. The effect of varying scenes on the recognition rate and false recognition is also considered.

Lighting condition	Scene Type	Avg. Time/Frame	False Recognition	Degree of Recognition
	Ideal	7.34	0	1
Indoor	Simple	10.89	0	0.964
	Complex	22.48	0.071	0.9367
	Similar	38.11	0.942	0.7877
Outdoor	Simple	15.4	0.01	0.93
	Complex	23.7	0.192	0.8545
	Similar	44.62	0.86	0.6999

Table 4.3: Performance of the algorithm for varying scene and lighting conditions

As can be seen from Table 4.3 the performance of the algorithm performs better for indoor scenes than outdoor scenes, since for the outdoor scenes there are glares and shadows that affect the performance. For scenes with similar background the average time per frame increases, since the number of false detections increases, increasing the regions to be included in the recognition phase.

Next the target recognition phase of the algorithm is tested when the size of the target varies. The test is taken for all the scenes, simple, complex and similar under both indoor and outdoor scenarios and the average of the tests is taken. As can be seen from the graph in Figure 4.2below, the degree of the recognition deteriorates as the object size grows. The size of the target as a percentage of the whole frame is taken in to consideration with respect to the maximum normalized recognition degree, found from the template matching result matrix.

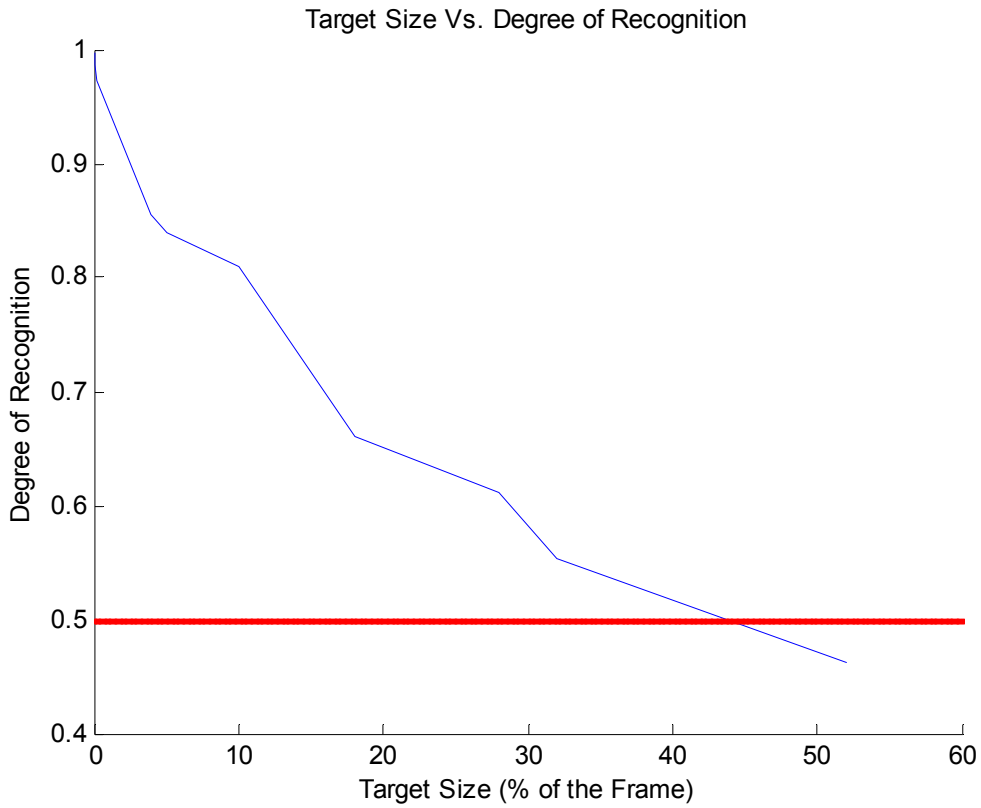

Figure 4.2: Target Template Size Vs. Recognition

Figure 4.2show the degree of recognition as the percentage of the target size to frame size varies. As the target size increases the degree of recognition decreases, since parts of the target can be cropped being out of the frame resulting in partial recognition. As can be

seen from the figure if target size exceeds 44% of the whole frame, the degree of recognition is below 0.5, resulting in misrecognition.

GPU Implementation Test

The first call of functions on GPU is quite slow; it took 9.38 seconds for our case. So for performance measuring, it is necessary to do dummy function call and only then perform time tests. We used $cv::gpu::getCudaEnabledDeviceCount()$ function for the first call and then performed the performance test. If it is critical for an application to run GPU code only once, it is possible to use a compilation cache which is persistent over multiple runs.

For GPU test videos with varying standard frame sizes has been tested, varying the frame sizes from Quarter Video Graphics Array (320 x 240) to HDTV Quad Extended Graphics Array (1920x1080).

Standard Name	Size	Aspect Ratio	Average Time/Frame(ms)		Speedup
			CPU	GPU	
QVGA	320 x 240	4:3	7.60	9.40	0.7862
HVGA	640 x 240	8:3	13.97	13.89	1.0058
VGA	640 x 480	4:3	24.83	15.33	1.6196
SVGA	800 x 600	4:3	50.67	25.40	1.9948
XGA	1024 x 768	4:3	69.50	34.67	2.0048
XGA+	1152 x 768	3:2	78.11	35.20	2.2189
	1152 x 864	4:3	86.44	35.61	2.4276
SXGA	1280 x 1024	5:4	94.50	47.33	1.9965
SXGA+	1400 x 1050	4:3	130.67	65.05	2.0086
UXGA	1600 x 1200	4:3	151.33	65.60	2.3069
QXGA	2048 x 1536	4:3	276.67	131.07	2.1109

Table 4.4: Speedup of the GPU implementation of ATR

As can be seen from the Table 4.4 the performance of the algorithm is better for the CPU-based implementation of the algorithm for QVGA videos. As the video size grows the performance of the algorithm gets better on GPU-based implementation of the algorithm. Figure 4.3 shows the speed up of the algorithm using GPU-based implementation. The speed up of the algorithm is below 1, which means for QVGA frame size the CPU-based implementation performance better than the GPU-based implementation. The speed up of the algorithm increases as the video size increases.

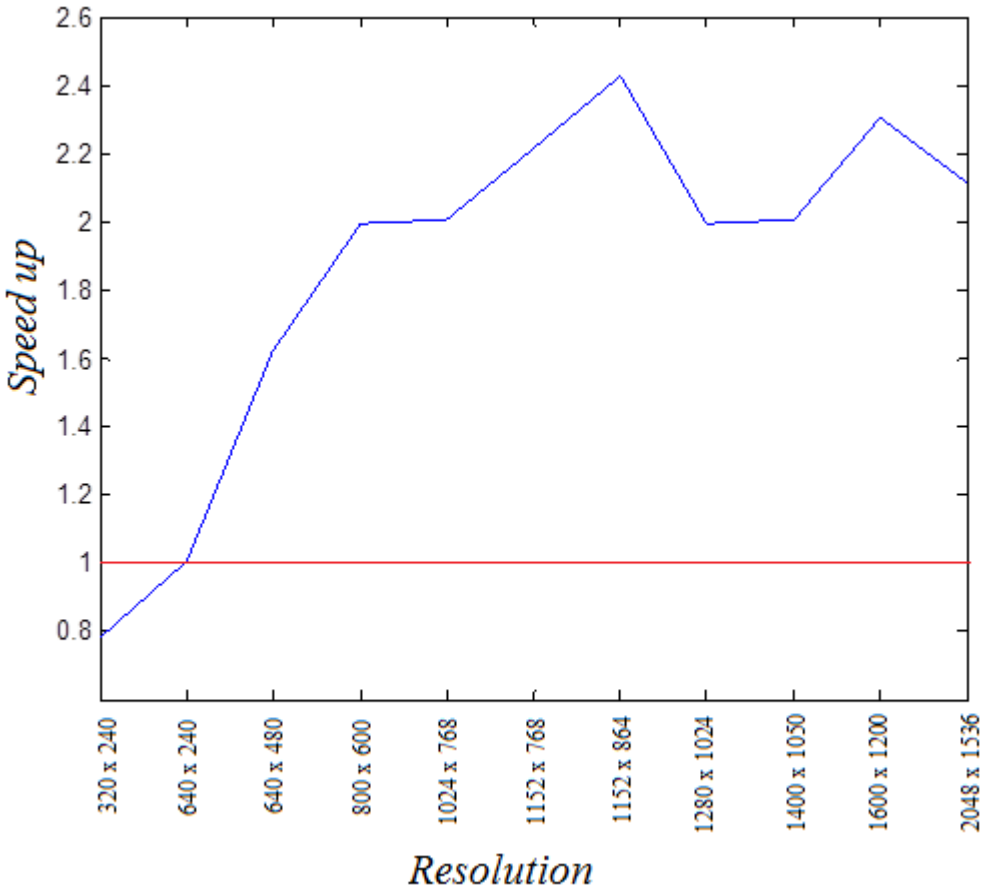

Figure 4.3: Speed up of the GPU-implementation over CPU-implementation

As can be seen from the graph in Figure 4.3 the GPU speedup obtained gets a pick value of 2.4 at the resolution of 1152 x 864. The highest speedup happens when the number of

threads required to process the frame fully occupies the number of processors available on the device. For bigger resolution the number of threads exceeds the number of parallel computations supported by the device so that some threads have to wait till processors finish their current task. But in the next round the number of threads executed is less than the parallel executions supported; the device is partially used, which decreases the obtained speedup.

4.4. Discussion

The performance of the algorithm varies significantly when the threshold value of the background subtraction phase is varied. Minimizing the threshold value improves the computation time, but the target miss rate is affected severely. On the other hand, increasing the threshold level improves the target detection miss rate but at the cost the computation time. We found the threshold level of 13 to be an acceptable with the performance not affected severely, with in a real-time frame rate region, and an acceptable level of target miss rate.

For recognition phase of the algorithm, the degree of recognition varies significantly with scene type and lighting conditions. For scenes where targets have the same color as the background the false detection rate increases, depreciating the computation time of the algorithm. Test has shown that targets with lower Hue range (narrow Hue range) have better results for similar colored targets and backgrounds.

For the GPU based implementation of the algorithm the initialization time required for the device makes it lagging for live target processing. To overcome this, the target processing has to be done on CPU at start and proceed on GPU after the initialization.

The GPU used for the experiments because of availability, GeForce 8400GS, is outdated compared to the CPU, Intel Core i3-2100. If GPU of equivalent technology and computation power as the CPU, such as GeForce 9800GS, were used an enormous speed up is expected. Table 4.5 shows the comparison of GeForce 8400GS, used for the experiments, and GeForce 9800GS.

Metric	GeForce 8400GS	GeForce 9800GS
Number of cores	16	112
Core speed	459MHz	600MHz
Memory speed	800MHz	1800MHz
Bandwidth	6.4GB/sec	57.6GB/sec

Table 4.5: GPU Comparison GeForce 8400GS vs. GeForce 9800GS [43]

The speed up gained from the GPU is due to the number of cores of the device. There is big difference in number of cores and memory speed between the two devices, which are the main reasons for an increased speed up.

CHAPTER 5

CONCLUSION AND SUGGESTIONS FOR FUTURE WORK

5.1. Conclusion

The work of this thesis demonstrates a performance improvement gained by using a hybrid algorithm for automatic target recognition. Both shape based and color based segmentation algorithms were used to get an improved performance and detection rate compared to each one of them separately.

The target recognition algorithm was tested both on CPU and GPU implementations. An appropriate choice of part of the algorithm is made for GPU implementation, with minimum data transfer. Computations where CPU is as fast as GPU remained at CPU as it appends unnecessary I/O transfer between CPU and GPU. Due to the inherent parallelism of the recognition algorithm, it is suited for GPU implementation. Since these is a component of the computation that work on huge data, Single Instruction multiple data units, exhibiting high degree of independence.

The performance of the system is tested for robustness and for variation in the properties of targets and the environment behind. The performance gain for using a co-processing on GPU has also been tested. A significant performance speed up is gained through the use of GPU.

5.2. Suggestions for future works

The need for performance improvement in multimedia applications computation has been and will be an ever increasing challenge in digital signal processing. Meanwhile the performance improvement of digital devices is increasing. Performance improvement of Computer vision algorithms can be achieved through the use upcoming digital devices by mapping the algorithm or part of the algorithm on the device and taking advantage of the nature of the specific device used.

The algorithm implemented in this work only considers 2-dimensional views, but with an improved performance in the devices, 3-dimensional views can be implemented with this algorithm. In addition a single camera configuration is considered for this work, future works with device improvements can be mapped to more than one integrated camera systems.

It is also worth noting that the accuracy of algorithm can be improved further by incorporating advanced detection methods like Steerable filters for edge detection and methods for prediction probability.

REFERENCES

[1] Ufuk Suat Aydin, Traffic Sign Recognition, Middle East Technical University, May 2009.

[2]Second LieutenantMichael A. Tanner, Image Processing forMultiple-Target Tracking on aGraphics Processing Unit, US Air Force, March 2009.

[3] Oscar Mateo Lozano · Kazuhiro Otsuka, Real-time Visual Tracker by Stream Processing, J Sign Process Syst. DOI 10.1007/s11265-008-0250-2

[4] Alok Whig, Stream Processor Based Real-Time Visual Tracking UsingAppearance Based Approach, University Of Florida, 2009

[5] Danish Nadeem, Saleha Rizvicharacter Recognition Using Template Matching, New Delhi

[6] Stephen Burns, Virtual Reality Simulation Using Stereoscopic Vision And Motion Tracking, California State University.

[7] Seiken Higashionna, Constructing A 3d Morphable Face From A Single Picture ByEstablishing A Correspondence From 2d Active AppearanceModel Parameters, University of North Carolina Wilmington, 2010

[8] Nicholas A. Vandal and Marios Savvides, CUDA Accelerated Iris Template Matching on Graphics ProcessingUnits (GPUs), 2010

[9] Sanyam Mehta, Arindam Misra, Ayush Singhal, et al. A high-performance parallel implementation of Sum of Absolute Differences algorithm for motion estimation using CUDA, 2009

[10] Dan E. Dudgeon and Richard T. Lacoss, An Overview ofAutomaticTarget Recognition, THE Lincoln Laboratorv Journa Volume 6, Number 1, 1993 .

[11] Bernd Jähne, Horst Haußecker,and Peter Geißler.Handbook ofComputer Visionand ApplicationsVolume 3: Systems and Applications,AcademicPress, 1999

[12] Zafer Savas, Real-Time Detection And Tracking Of Human Eyes In VideoSequences, Middle East Technical University, August 2005.

[13] Samet Karakaş, Detecting And Tracking Moving Objects With An Active Camera In Real Time, Middle East Technical University, September 2011.

[14] Johan A. Huisman, High-speed parallel processing on CUDA-enabled Graphics Processing Units, Delft University of Technology, June 7, 2010.

[15] NVIDIA, CUDA Compute Unified Device Architecture,Programming Guide, Version 1.1, November 29, 2007.

[19]F. Meyer, "Color image segmentation," Proceedings of the International Conference on Image Processing and Its Applications (pp. 303–306), 1992.

[20] Amdahl, G. "The validity of the single processor approach to achieving large-scale computing capabilities". Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, 1967, pp. 483–85.

[21] John L. Gustafson, "Reevaluating Amdahl's Law", Communications of the ACM 31(5), 1988, pp. 532–33

[22]The CUDACompiler DriverNVCC,Last modified on: <04-01-2008>

[23] C. Vachier, F. Meyer,The viscous watershed transform, Journal of Mathematical Imaging and Vision, volume 22, issue 2-3, May 2005.

[24] C. Rother, V. Kolmogorov and A. Blake, GrabCut: Interactive ForegroundExtraction using Iterated Graph Cuts in ACM Transactions on Graphics (SIGGRAPH) volume 23, issue 3, August 2004.

[25] S. K. Pal et al., "A Review of Image Segmentation Techniques", Pattern Recognition Conference, 29, 1277 – 1294, 1993.

[26] H. D. Cheng, X. H. Jiang, Y. Sun and Jing Li WangColor Image Segmentation: Advances and Prospects, Utah State University, Logan, UT 84322 – 4205.

[27] J. A. Gartaganis and J. Tartar, Wavebased Spectrum Rendering, Technical Report on Conference of Spectrum Analysis, 1990.

[28] Multi-Resolution Color Image Segementation Jianqing Liu, January 1993

[29] D. Comaniciu and P. Meer, "Mean shift analysis and applications," IEEEInternational Conference on Computer Vision (vol. 2, p. 1197), 1999.

[30] OpenCV | Willow Garage, 2008. Real time computer vision [online] Available at: <http://opencv.willowgarage.com> [Accessed 10 October 2012].

[31] Ohloh | cvBlob, 2012. Binary image labeling [online] Available at: <http://www.ohloh.net/p/cvblob> [Accessed 10 October 2012].

[32] CMAKE | Build system, 1999. Build, test and package software. [online] Available at: <http://www.cmake.org> [Accessed June 4 2012].

[33] X. Wan and C.-C. J. Kuo. Color distribution analysis andquantization for image retrieval. In SPIE proceedings, vol.2670, February, 1996.

[34] Scott Hemmert, Brad Hutchings and Anshul Malvi. An Application-Specific Compiler for High-Speed Binary Image Morphology, Brigham Young University.

[35] Sajid, M. M. Ahmed, I. Taj, M. Humayun, and F. Hameed, Design of High Performance FPGA Based Face Recognition System, Mohammad Ali Jinnah University, Islamabad, Pakistan.

[36] Xiaohan Chen and Natalia A. Schmid, On the Limits of Target Recognition in the Presence ofAtmospheric Effects. West Virginia University, Morgantown, WV 26506

[37] Tien-Hsin Chao and Thomas Lu, Automatic Target Recognition (ATR) Performance ImprovementUsing integrated Grayscale Optical Correlatorand Neural Network, California Institute of Technology

[38] Young H.Cho, Optimized Automatic Target Recognition Algorithm on Scalable Myrinet/FieldProgrammable Array Nodes,The University of Texas at Austin, U.S.A.

[39] Michael Rencher and Brad L. Hutchings, Automated Target Recognition on Splash. Brigham Young University, Provo, UT 84602

[40] Mahsa Maghami,Michael C. Koval.Social Network Analysis for Automatic Target Recognition in SwarmRobotics, University of RochesterRochester, NY, USA

[41] Jue Wang, Bo Thiesson, Yingqing Xu, Michael Cohen. Image and Video Segmentation by AnisotropicKernel Mean Shift, Microsoft Research (Asia and Redmond).

[42] Darrin Cardani, Adventures in HSV Space, Adobe Systems, Inc.

[43]Products | NVIDIA, GeForce. [online] Available at: <http://www.nvidia.com> [Accessed 03December 2012].

APPENDICES

Appendix A:Code used for speed up comparison of Cvblob and cv::findcontours

```
// Cvblob Vs. cv::findcontours
#include "cv.h"
#include "highgui.h"
#include "BlobResult.h"
int main()
{
   CBlobResult blobs;
   CBlob *currentBlob;
   CvPoint pt1, pt2;
   CvRect cvRect;
   int key = 0;
   IplImage* frame = 0;
  IplImage* image = cvLoadImage( filelocation );
 // Can't get device? Complain and quit
        if( !image)
         {
                printf( "Could not initialize capturing...\n" );
                return -1;
         }
        cvNamedWindow( "Image" );
        cvNamedWindow( "Blobs" );
        cvNamedWindow( "Contours" );
```

```
        if( !( frame = image ) )
            break;
double durationb,durationc;
durationb = static_cast<double>(cv::getTickCount());
        // using Blob
        blobs = CBlobResult( image, NULL, 0 );
        // Attach a bounding rectangle for each blob discovered
        int num_blobs = blobs.GetNumBlobs();


            for ( int i = 0; i < num_blobs; i++ )
            {
                    currentBlob = blobs.GetBlob( i );
                    cvRect = currentBlob->GetBoundingBox();
                     pt1.x = cvRect.x;
                    pt1.y = cvRect.y;
                    pt2.x = cvRect.x + cvRect.width;
                    pt2.y = cvRect.y + cvRect.height;
                    // Attach bounding rect to blob in orginal video input
                    cvRectangle( frame,   pt1, pt2,  cvScalar(0, 0, 0, 0), 1,  8, 0 );
            }
            durationb = static_cast<double>(cv::getTickCount())-durationb;
            cvShowImage( "Image", image);
            cvShowImage( "Done", frame );
            durationc = static_cast<double>(cv::getTickCount());
             // using cvfind contours
```

```
        CvSeq* contours = 0;
        CvMemStorage* g_storage = NULL;
        g_storage = cvCreateMemStorage(0);
        cvFindContours( image, g_storage, &contours );
        if( contours )
            cvDrawContours(image, contours, cvScalarAll(255), cvScalarAll(255), 100);
        cvShowImage( “Contours”, image );
         durationc = static_cast<double>(cv::getTickCount())-durationb;
        durationb /= cv::getTickFrequency(); // the elapsed time in ms
        durationc /= cv::getTickFrequency(); // the elapsed time in ms
        std::cout<<"Cvblob:"<<durationb<<" Findcontour:"<<durationc<<std::endl;
        cvReleaseImage( &imgage );
        return 0;
}
```