GPU Accelerated Parallel Iris Localization

Abhishek Sinha

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela – 769 008, India

GPU Accelerated Parallel Iris Localization

Dissertation submitted in

May 2013

to the department of

Computer Science and Engineering

of

National Institute of Technology Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Abhishek Sinha

(Roll 211CS1055)

under the supervision of

Prof. Banshidhar Majhi

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Dedicated to my Parents

Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, India. www.nitrkl.ac.in

Dr. Banshidhar Majhi
Professor

June 3, 2013

Certificate

This is to certify that the work in the thesis entitled *GPU Accelerated Parallel Iris Localization* by *Abhishek Sinha*, bearing roll number 211CS1055, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Banshidhar Majhi

Acknowledgment

I would like to take this opportunity to express sincere gratitude towards my thesis supervisor, Prof. Banshidhar Majhi, who has been a constant source of encouragement throughout the course of this thesis work. I am indebted to him for the valuable advice he has given me for choosing the area of work, and carrying it forward. Without his advice the completion of the thesis would not have been possible.

I thank the head of the department of Computer Science and Engineering, Prof. Ashok Kumar Turuk and all other professors for providing the resources and the environment needed for the successful completion of the thesis. I thank the admin and staff of National Institute of Technology Rourkela for extending their support whenever needed.

I also thank my friends and peers who have extended their help whenever needed. Their contribution has always been significant. Finally, I would like to take this opportunity to thank my parents, who have always been a source of inspiration and motivation for me, and also for the love they have provided in stressful periods, which has been a guiding force for the completion of the thesis.

Abhishek Sinha

Abstract

Iris recognition is quite a computation intensive task with huge amounts of pixel processing. After the image acquisition of the eye, Iris recognition is basically divided into Iris localization, Feature Extraction and Matching steps. Each of these tasks involves a lot of processing. It thus becomes essential to improve the performance of each step to gain an overall increase in performance.

The localization step is of utmost importance since it finds out the essential region over which further steps of Iris Recognition are to be performed. It thus decreases the amount of computation that will be needed in the subsequent steps.

In this thesis an effort has been made to improve the performance of Iris localization by the use of parallel computing techniques. Recently the General Purpose Graphics Processing Units(GPUs) have come to be very popular in solving complex computational tasks. In order to achieve a speedup in the localization step, the Compute Unified Device Architecture(CUDA) platform released by NVIDIA corporation has been used.

Hough Transform for circles has been used to perform the localization step since it has the ability to handle noisy data very efficiently. The edge image has been obtained using the popular canny edge detector and it serves as the input for the Hough Transformation step. Since the image data as well as the edge detecting mechanism may not be perfect, the Hough transform method carries out a voting mechanism over the image objects, in order to deal with imperfections like noisy data.

Parallelism is employed in the Hough transformation step, when for each possible value of the radius a large number of circles have to be generated in the parameter space, and this task is taken over by parallel blocks and threads, which substantially improves the computation time required to identify the circular contours in the image space.

***Keywords*:** GPU, CUDA, Thread, Block, Kernel, Hough Transform, Canny Edge Detector,

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Parallel computing has been an area of active research during the past few years, with a lot of technical enthusiasts trying to improve the efficiency of applications through parallel computing techniques and algorithms. It has been applied to a number of fields ranging from weather forecasting to mineral discovery, and image processing is no exception. All these applications involve a heavy amount of computation. With the recent growth in the storage and computing power of machines, the demand for faster processing has increased like never before.

Recent trends have shown tremendous improvement in the growth of parallel computing especially in the area of image processing, video processing, graphics rendering and visualization. The advanced graphics cards today are capable of rendering graphics applications flawlessly unlike it was the case a few years ago. Current graphics cards have tens or even hundreds of processing cores capable of high precision floating point arithmetic.

1.1 Basics of Parallel Computing

Parallel computing aims to solve a complex problem by dividing the problem into subsets and solving each problem as a separate thread that can be executed in parallel [1]. Parallel algorithms tend to utilize the resources of the system most

effectively taking the advantage of the multicore systems which are in widespread use today. It aims to explore the concurrency that can be achieved while the execution of an application. Most of the real time applications tend to be computing intensive. Hence it becomes essential to explore the parallelism in the applications, if any, which provides significant speedup to the application.

1.1.1 Speedup

The gain of a parallel algorithm is expressed in terms of speedup which is defined as the ratio of the time taken to execute the application on a serial machine to the time required to execute the application on a parallel machine [1].

$$Speedup = \frac{T_s}{T_p} \tag{1.1}$$

T_s is the time taken to execute the application on a serial machine and T_p is the time taken to execute the application on a parallel machine.

This speedup is affected by a host of factors and overheads. The following are the limitations to the achieved speedup of a parallel machine:

Communication Latency

The sequential problem is decomposed and mapped to n number of processors, with each processor executing on a particular subset of the dataset. During parallel computation the communication involved between the processing elements becomes the limiting factor for the gain in speedup. Considerable attention has to be paid in handling the amount of communication involved among the processors as well as the amount of communication involved between a processor and the memory.

Amount of Sequential Code

Parallel algorithms are also limited by the amount of sequential code present in the algorithm. If the sequential code present in an algorithm takes larger execution time than the parallel code, the benefits of the parallel code are lost. As an example, if 25 percent of an application is serial then any number of cores can't provide a speedup of more than 4 times. Therefore careful exercise has to be conducted before parallelizing the application otherwise the resources simply go into vain.

Task Dependency

Another limitation in the application of parallel computing is the dependency in the tasks. If the tasks are dependent on previous tasks it becomes difficult to employ parallelism on these tasks since the results of the previous tasks must be available in order to accomplish the new tasks. So parallelism cannot be carried out. Parallelism is most beneficial when the divided tasks are virtually independent of each other and can be carried out simultaneously, without the results of one affecting the other. In the end the results can be simply combined to obtain the final result. It requires considerable programming effort to accomplish independent parallel execution of the tasks.

1.1.2 Classification of Parallel Systems

The parallel computation frameworks are generally modeled based on Flynn's classification [2] who divided the parallel architecture into four groups :

(i) SISD (Single Instruction Single Data) : This group of computing systems consists of a single sequential processor which executes the tasks one after another thus employing no parallelism.

(ii) SIMD (Single Instruction Multiple Data) : This is the most important class of parallel architecture which employs multiple processing elements which operate

on different sets of data. The noticing factor is that all the processing elements run the same algorithm to employ the same operation on the data sets. Thus this architecture comes most handy when similar operation is to be applied to different sets of data.

(iii) MISD (Multiple Instruction Single Data) : This version of computer architecture applies multiple instructions on a single set of data on different processors. Generally this kind of feature is of little use when it comes to parallel computation.

(iv) MIMD(Multiple Instruction Multiple Data) : This architecture is also of utmost practical use when different algorithms have to be run on different sets of data. This kind of architectures are mostly found in distributed environments.

SIMD machines thus employ data level parallelism. Even modern GPUs are wide SIMD implementations capable of branches, loads and stores 128 to 256 bits at a time. SIMD instructions are widely used to process 3D graphics and modern graphics cards with embedded SIMD have largely taken over this task from the CPU.

1.1.3 Communication Methodologies

The transfer of data among the parallel cores is the backbone of parallel processing, since in most of the cases it is required to transfer data from one core to another and this is even more frequent if the elements of data are interdependent. The efficiency of a parallel algorithm depends on, and is in fact limited by the time taken to communicate different pieces of data between the processing elements. Sometimes it also happens that the time taken for the communication of data between the processing elements becomes more than the time required for the parallel calculation of the results. In such a case the benefits of parallel algorithm will be totally lost. It thus becomes necessary for the design of suitable techniques for the communication of data and optimization of the amount of data flowing between the processing cores.

Two techniques exist for the communication of data between the different processing elements [1] :

Shared Address Space Mechanism

In this method a common memory is accessible to all the processing cores. The shared memory address space may be uniformly or non-uniformly accessed. In Uniform Memory Access (UMA), the time taken by a processor to access any memory location is the same whereas in Non Uniform Memory Access (NUMA), the time taken by a processor to access different memory locations may be different. Thus each memory has equal privilege to access any memory location.

Message Passing Platform

In this mechanism each processor has its own memory and address space. The interaction among the processing cores is achieved through the transfer of messages among the processing cores. Message passing Interfaces are used to support such interactions. The exchange of messages is used to transfer data, work, and to synchronize actions among the processes. In its most general form , message passing paradigms support execution of different program on each of its p nodes. The basic operations in this programming paradigm are send and receive messages. Also there must be a method to assign unique identification to each of the processes running a parallel application.

1.2 Introduction to Iris Localization

Iris recognition is an automated method of biometric identification that uses mathematical pattern-recognition techniques on the images of the irises of an individual's eyes, whose complex random patterns are unique and can be seen from some distance. The benefit of using Iris as a biometric trait is that Iris is a protected internal organ and its random texture pattern remains unchanged throughout the lifetime of an individual. A sample image of an iris is shown in figure 1.1.

Iris recognition uses camera technology with subtle infrared illumination to

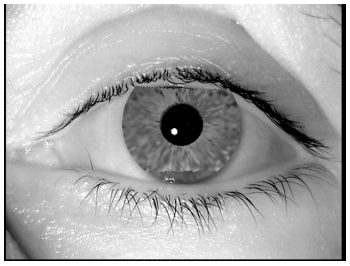

Figure 1.1: A Sample iris image

acquire images of the detail-rich, intricate structures of the iris. Digital templates encoded from these patterns by mathematical and statistical algorithms allow the identification of an individual. Databases of enrolled templates are searched by matcher engines at speeds measured in the millions of templates per second per CPU, and with infinitesimally small False Match rates.

Many millions of persons in several countries around the world have been enrolled in iris recognition systems, for convenience purposes such as passport-free automated border-crossings, and some national ID systems based on this technology are being deployed. A key advantage of iris recognition, besides its speed of matching and its extreme resistance to False Matches, is the stability of the iris as an internal, protected, yet externally visible organ of the eye.

Iris is very popular as a biometric trait because it is potentially invariant to age, psychological condition and a slew of other external factors. It is used to identify people on national and international level databases, as a security measure for biometric based authentication and verification. Other biometric traits like signatures and speech are affected a lot by psychological factors, and here Iris biometric comes to rescue whose random texture remains stable throughout the lifetime of the individual. Moreover it is a protected organ free from external damage, thus more reliable.

Currently, iris recognition algorithms are deployed globally in a variety of systems ranging from personal computers to portable iris scanners. These systems use central processing unit based systems for which the algorithm was originally designed.

CPU-based systems are considered general purpose machines, designed for all types of applications. Therefore, CPU-based systems are known as sequential processing devices. Instructions are first fetched, decoded, and finally executed by arithmetic logic units, or ALUs. State-of-the-art processors contain more than one ALU. Therefore, multiple instructions can be executed in parallel. However, modern processors are limited in the number of ALUs they possess, and most of today's CPUs do not have more than four processors so utilizing the GPUs could be an alternative solution to accelerate the application .

Iris biometric identification is essentially a pattern recognition work involving the conventional steps of preprocessing, feature extraction, feature representation, and identification. All these steps involve considerable amount of computation complexity, the efficiency of which can be drastically improved using parallel algorithms and architecture. This becomes even more important when the identification or authentication process has to be done over national level databases. National level databases may contain millions of templates to match, and each of them involves huge amount of pixel processing, so parallel computation is the only alternative.

The modern day GPUs are not only used for Graphics rendering and visualization but also for dealing with other general purpose calculations requiring heavy amount of computation, hence they are sometimes also called as General Purpose graphics Processing Units or GPGPUs. These GPGPUs come very handy for research purposes, when the general algorithm takes large amount of time and getting quick result becomes impossible.

The process of Iris recognition is basically divided into the steps of Image acquisition, Iris Localization, Template generation and Matching as shown in figure 1.2.

The overall performance of an Iris recognition algorithm greatly depends on the performance of the localization algorithm which forms the initial step in the process for detecting a match for the acquired image against the templates available in the

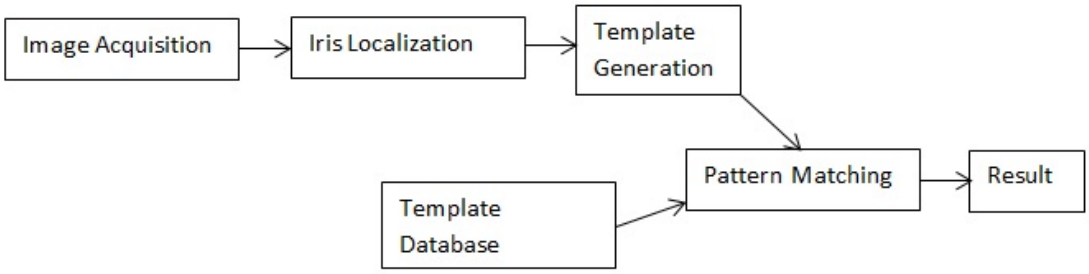

Figure 1.2: Steps involved in Iris Recognition

database. Template generation consists of generating a feature vector unique to the image, by analyzing the texture patterns available in the Iris image. The step of Iris Localization involves finding the inner and outer boundaries of the Iris in order to separate the regions of the acquired image which are not relevant for further processing. A sample image and the localized image of an iris are shown in figure 1.3. The relevant portion of the image are those that lie outside the pupil and inside the outer boundary of the Iris. Also the region of the Iris occluded by the eyelids should be removed.

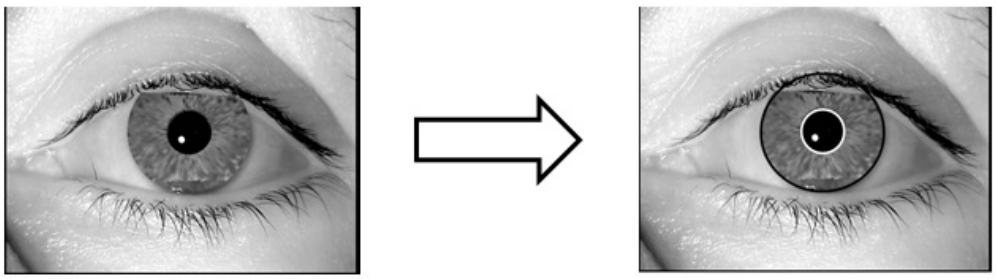

Figure 1.3: Results of Iris Localization without the removal of the portion occluded by the eyelids

1.3 Motivation

There have been significant efforts to develop parallel algorithms for the purpose of Iris recognition, and it is still an area of active research, and that has been a source of motivation for the project. Another source of motivation has been the recent advancements in the growth of Graphics Processing Units, which have been widely used for research projects in order to accelerate applications, which require substantial amount of time on conventional systems. Since iris recognition is a computation intensive image processing application, it serves as a good candidate for the application of parallel processing. Localization is an important part of iris recognition process, hence application of parallel processing to speedup the process improves the overall process of iris recognition.

1.4 Problem Statement

The initial observation to the process of Iris recognition has been the fact that each stage of the process deals with good amount of processing, and thus takes considerable amount of time. Up to now, most of the methods developed for iris recognition have been sequential in nature. This factor coupled with the growth of parallel infrastructures like the GPUs has been a tempting factor for this thesis.

The localization of the iris is an important step in the process of iris detection and it takes considerable amount of time in its sequential implementation. The localization process has been accelerated by the use of parallel hough transform, which is a popular and robust method to detect the presence of shapes in an image.

1.5 Thesis organization

The remaining part of the thesis is organized as follows :

Chapter 2 provides a basic introduction to GPU Architecture and CUDA programming elements to understand the thesis better. This chapter is intended

to help the reader understand the further parallel algorithms defined in CUDA.
Chapter 3 deals with the important works done in the field of Iris recognition so far and the serial and parallel approaches taken toward the process of Iris localization. The recent parallel works done in iris recognition have also been explained.
Chapter 4 describes the serial Hough transform method and the proposed parallelization of the method for the purpose of Iris localization.
Chapter 5 describes the implementation of the algorithm in CUDA and the obtained results for iris localization.
Chapter 6 provides the concluding remarks.

Chapter 2

GPU Architecture and CUDA

The tremendous growth in the advancement of computer graphics has resulted into the evolution of high performance graphics cards which are capable of high precision floating point arithmetic and visual rendering. Graphics cards are used for rapid creation of frames for visual display, and are generally used in Personal computers, Mobile Phones and Gaming Consoles. The creation of these frames is highly computation intensive according to modern day display requirements especially in computer games where the next frame to be displayed depends on the complex real time interaction of the user with the game and also in video display where complex encoding and decoding logics are in play. The term Graphics Processing Unit (GPU) was coined by NVIDIA Corporation which used the term to popularize its graphics cards.

Modern day graphics cards are so computationally powerful that it soon grabbed the attention of researchers in various fields trying to accelerate their applications for getting quick results to their research endeavors. Such cards are also sometimes referred to as General Purpose Graphics Processing Units (GPGPUs). It also led to various researchers trying to accelerate their applications in order to increase the efficiency of their algorithm, by creating a GPU version of their algorithm, and the field of image processing is no exception. Many researchers have come out with the GPU versions of their algorithms and have received great appreciation.

2.1 GPU Architecture

The GPUs are highly parallel, multithreaded programmable devices capable of rendering realtime graphics applications. It has tens, hundreds or sometimes even thousands of cores with tremendous power capable of high precision floating point arithmetic which is the requirement of realtime video processing. Another important feature of the graphics cards is that it has a very high memory bandwidth of at least 64 bits in the modern processors and a large number of on chip registers. The high bandwidth allows large amounts of data transfers in a single flow while the large number of registers allows it to hold several variable values while computation [23].

The number of floating point operations per second(FLOPS) for the GPU is exponentially higher compared to that of a high end CPU. A high end CPU has a computation speed of a few tens of gigaflops whereas even a normal GPU has a computation speed of few hundred gigaflops. High end GPUs have computation speeds in few teraflops. The reason for this is that GPUs are highly optimized for performing compute intensive highly parallel tasks such as graphics rendering. It is thus designed such that large number of transistors are devoted to data processing rather than flow control and data caching.

The GPUs are suitable for problems which can be expressed as a data parallel application, i.e., the same program is used on different sets of data and the arithmetic operations are much higher than memory operations. Since the same program is executed on different data sets on different processing elements, there is very less requirement for the flow of data from one processor to another. The memory latency is therefore not visible and it remains hidden under the heavy calculations that take place inside the processors [23].

Any process that uses large data sets can use a parallel programming model to speed up the application. In 3D image rendering applications like computer gaming programs, large number of pixels are mapped onto processing cores for independent parallel processing. Similarly video encoding and decoding applications can map a subset of data like an image block onto a separate processor for execution [23].

2.1.1 GPU Memory Organization

Like a normal motherboard a GPU is a printed circuit board that contains a processor and a RAM. It also has a BIOS typically optimized for the initial settings of the GPU. The RAM also acts as a frame buffer which means that it holds the computed frame into memory before displaying it. The computation of each frame of a realtime application is highly compute intensive.

The NVIDIA GPUs are partitioned into a set of Streaming Multiprocessors(SMs) as shown in figure 2.1. A Single Streaming Multiprocessor generally consists of 8 cores. Thus if a GPU has 48 cores than it is generally organized into 6 Streaming Multiprocessors each with 8 cores. Anyway it may vary depending on the configuration of the GPU and may contain higher number of processors per Streaming Multiprocessor if the GPU has a large number of cores. The programs that run on the GPU have nothing to do with the architectural differences, and this makes programming of the GPU scalable.

The Streaming Multiprocessor schedules the instructions across the cores. Upto 32 threads can be scheduled at once on a Streaming Multiprocessor known as warp scheduling. Each core in a SM contains its own register level memory which is available to a single thread for its execution on the core. In addition there is a shared memory for the Streaming Multiprocessor which is available to all the threads getting executed on the SM. All the threads running on a given SM, which need to cooperate, use this piece of memory for communication.

Each SM executes a group of cooperating threads. Also there is a global memory commonly known as GPU memory which is available to the entire GPU and is generally used to hold the data copied from the host(CPU) to the device(GPU) as well the intermediate results before they are transferred to the CPU. In video processing, this memory holds the next frame to be displayed after computation. This memory is similar to the Random Access Memory (RAM) available for the CPU to use. All the communication between the CPU and the GPU takes place through this memory.

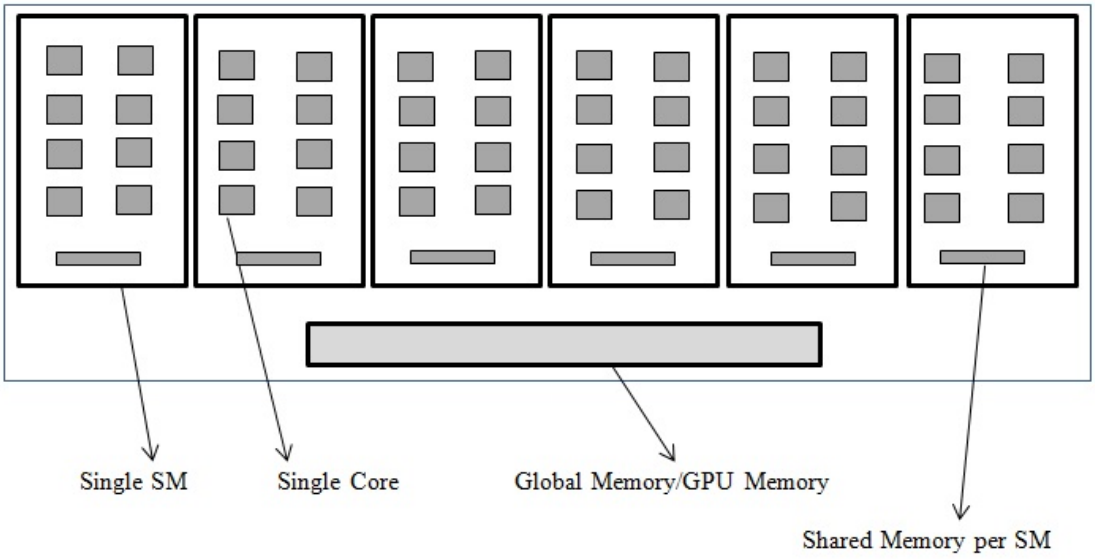

Figure 2.1: Architecture of a GPU with 48 cores, 8 cores per SM

2.1.2 Limitation

The limitation in the use of a GPU as a GPGPU is that it can process only independent fragments of data. The benefit is that many such fragments can be processed in parallel. GPUs are thus known as stream processors since they can run the same operation on multiple streams of data and thus can be used only for SIMD operations.

2.2 CUDA Elements

Compute Unified Device Architecture or CUDA for short is a proprietary technology introduced by NVIDIA corporation in 2006 for the first time. CUDA is a parallel computing platform and programming model which uses the Parallel Compute Engine to manage the threads running on the GPU. It is the task of the Parallel Compute Engine to schedule the threads to be run on the GPU and map them to the available cores such that it conforms to the CUDA architecture and specification [23].

CUDA platform architecture is designed in such a way that it can support

multiple languages and APIs. CUDA provides support for high level languages like C, C++, FORTRAN, DirectCompute and OpenACC, while third party wrappers are available for other languages like Python and Java [23].

2.2.1 Building Blocks

A thread is the smallest unit of a CUDA program and it serves as the building block of a CUDA program. A group of threads is termed as a block and in turn a group of blocks is known as a grid. It is the task of the programmer to specify the number of blocks to be created for running the parallel application. The programmer also specifies the maximum number of threads that can be created per block.

$$Number\,of\,Blocks\ =\ \frac{Total\,Number\,of\,Threads}{Number\,of\,Threads\,per\,Block} \tag{2.1}$$

There is a limitation to the number of threads that can be included per block. This is due to the limited amount of shared memory that is available for a SM. In the current line of GPUs the maximum number of threads that can be created per block to be executed on a single SM is 1024. Since the cooperating threads use this shared memory this limitation becomes obvious.

The CUDA Compute Engine maps the blocks to the available Streaming Multiprocessors. There is another restriction to this. All the threads within a block must be scheduled to run on the same Streaming Multiprocessor. If the number of threads in a block is higher than the total number of processors on a SM, then in the first pass only as many threads are scheduled as the number of processors on the SM, while the remaining threads wait for the current batch of threads to complete its execution. In this manner all the threads complete its execution [23].

2.2.2 Scalability of a CUDA application

CUDA programs are highly scalable in the sense that the programmer does not have the need to think about optimizing the program for the GPU he is using. The same application will successfully run on any GPU with any number of cores. It is the sole responsibility of the GPU Compute Engine to schedule the threads appropriately on the device with a given architecture. Figure 2.2 shows the mapping of eight thread blocks on two different GPUs, one with 2 SMs and another with 4 SMs.

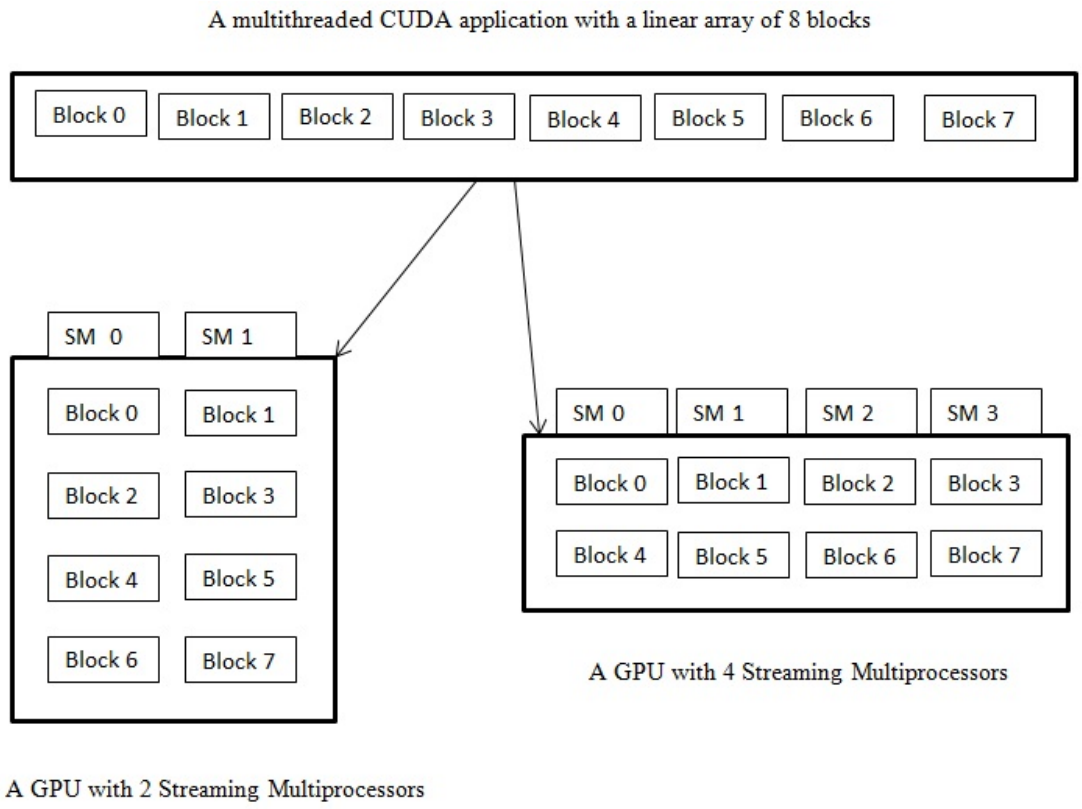

Figure 2.2: Scalability of a CUDA Application

The scalability of a CUDA program allows the same application to be run on a variety of NVIDIA machines from the general purpose GeForce GPUs to the professional Quadro and Tesla products. This is done by simply scaling the number of multiprocessors and memory partitions. Thus a GPU with more number of

multiprocessors will automatically run the program in less time compared to a GPU with less number of multiprocessors.

2.2.3 Elements of a CUDA application

As discussed before, CUDA allows the use of several high level languages to develop a parallel application. When C is used, CUDA programs are essentially C programs with a set of APIs and library functions, but the programming syntax remains same as that of a C program. CUDA programs are built around special functions called as Kernels. When these kernel functions are called, they are executed N number of times in parallel. Each of the N executions of the kernel is carried out by a separate thread, which may belong to one of the blocks created by the program scheduled to run on one of the Streaming Multiprocessors [23, 24].

A CUDA kernel is defined as follows:

__global__ void kernel_function_name (parameter list)

The keyword __global__ indicates that the code runs on the device and is called from the host code. Every thread that executes the kernel is given a unique identification ID called threadIdx. This value is available from within the kernel. The threadIdx variable is a three component value so that threads can be arranged along a single dimensional array or an array of two or three dimensions. For a single dimensional array arrangement , a thread is identified by threadIdx.x. For a 2d array arrangement it is identified by threadIdx.x and threadIdx.y. For a 3D arrangement the index of a thread is calculated by using threadIdx.x, threadIdx.y and threadIdx.z. This helps in simplifying the program logic for dealing with a vector, matrix or volume [23].

CUDA arranges the threads and blocks in the form of 1D, 2D or 3D array. Just as the organization of threads is done, in the same manner blocks are arranged

into a one-dimensional, two-dimensional or three-dimensional grid of thread blocks. And similarly the blocks within a grid are identified by a three component vector specified by blockIdx value, which has three components blockIdx,x, blockIdx.y and blockIdx.z.

The dimension of a thread block, i.e. the number of threads per block is available in a predefined variable blockDim a three component value. The syntax to call a CUDA kernel, also known as a kernel launch, is as follows:

kernel_function_name<<<NUM_BLOCKS, THREADS_PER_BLOCK>>>(parameter list);

The variable NUM_BLOCKS specifies the total number of blocks to be created for the application while the variable THREADS_PER_BLOCK specifies the total number of threads to be scheduled per block. In CUDA programming the CPU and the RAM is called as the host and the GPU is called the device. Any CUDA program consists of the following steps :

- Host memory allocation for the required variables
- Device Memory allocation for the variables
- Copying the values from host to device
- Computation of the results on the device by the use of parallel threads executing with the help of CUDA kernels.
- Copying the computed values back to the host

Chapter 3

Literature Review

The concept of automated Iris recognition was originally put forward by Leonard Flom and Aran Safir [3] in 1987, both ophthalmology professors, for which they subsequently obtained a patent entitled "Iris Recognition Technology". They then approached John Daugman [4–7], a professor at Harvard University, to write an algorithm based on their concept. Daugman constructed an algorithm, for which he obtained a patent few years later. After the patent expired, several other algorithms based on Daugman's method were created, some of which had a performance much better than Daugman's algorithm.

3.1 Iris Recognition

The Iris of the human eye contains complex random texture patterns which can be extracted from the digital image of the human eye, and encoded into a template, using image processing techniques. This template can be stored in a database. The template contains a mathematical representation of the unique information stored in the iris, and this allows the comparison of templates from the database.

The first operational Iris recognition system was developed by Daugman at the University of Cambridge in 1994. Daugman used multi-scale quadrature 2D Gabor wavelets to extract texture information in order to generate a 2048 bit pattern called

IrisCode [4]. All the commercial systems in operation today are based on his method.

In summary Daugman's method is implemented through :

- An Integro differential operator to detect the inner and outer boundaries
- Multi-scale quadrature 2D Gabor wavelets to extract texture information in form of unique binary vectors constituting the IrisCode
- A statistical matcher (logical XOR operator) which finds out the hamming distance between the input IrisCode and the IrisCodes in the database

In another important work following Daugman's work, Wildes et al. [8] have represented the texture present in the Iris by constructing laplacian pyramids with different resolution values. They have used normalized correlation to find out whether the input image and the model image are from the same class. R.Wilde's solution comprises of the following steps :

- A Hough Transform for Iris localization
- Laplacian pyramid(multi-scale decomposition) to represent distinctive spatial characteristics of the human iris, and
- Modified normalized correlation process for the matching step.

3.2 Iris Localization

Iris localization forms the first stage of the iris recognition process, in order to isolate the actual region of interest present in the eye. Normally the eyelids and eyelashes are present as the occluding parts of the iris region. Figure 3.1 shows an image of a localized iris obtaind after the removal of eyelids and eyelashes.

The original Iris recognition algorithms were essentially serial in nature, i.e. meant to be deployed on traditional CPU based systems. The work started with Daugman, and it has taken several flavors till recently. According to Daugman's

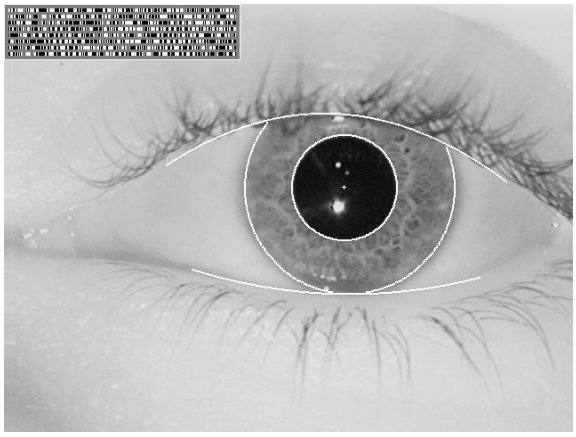

Figure 3.1: Localization after removal of eyelids

methodology, Iris recognition consists of three steps. First step consists of the acquisition of the eye image using a digital video camera. Near infrared illumination is used for image acquisition so that a darker iris will show more details [4]. Then the step of iris localization is carried out which extracts the iris from the raw image. This technique is shown in subsequent sections, and some other methods have also been discussed. Then Gabor wavelets are used for feature extraction to generate 2048 bit binary code called IrisCode. Then finally matching step is performed.

3.2.1 Daugman's Method

For the purpose of Iris localization, Daugman suggested an integro-differential operator that searches for the maximum in the blurred partial derivative of the image, with respect to the radius of the normalized contour integral of the image, along a circular arc of radius r and center (x_0,y_0) [6, 7].

$$max(r, x_0, y_0) \mid G_\sigma(r) * \frac{\partial}{\partial r} \int_{(r,x_0,y_0)} \frac{I(x,y)}{2\pi r} ds \mid \tag{3.1}$$

I(x,y) is the image and $G_\sigma(r)$ is the smoothing kernel. Different values σ provide different levels of smoothing. The fact that the pupil is not always darker than

the iris is addressed by taking the absolute value of the partial derivative. The operator in totality behaves as a circular edge detector. Some other segmentation methods [10–13] have also been very popular.

The recognition process of the Iris does not consider the color of the iris, though there is a significant variation in the color of the iris ranging from black to brown as well as bluish to greenish, rather the method focuses on the texture patterns like the furrows, crypts, rings and corona. After localization, a 2048 bit code for the Iris is calculated and then compared to the templates present in the database. The patterns are encoded using 2D Gabor demodulation. The IrisCode consists of 2048 bits of data plus 2048 masking bits producing an IrisCode of 512 bytes.

3.2.2 Wilde et al.'s Scheme

In Wilde's scheme [8, 26], first the image intensity information is converted into a binary edge map. Then the edge points vote to instantiate particular contour parameter values. The edge map is obtained using a gradient based edge detection scheme.This is done by thresholding the magnitude of the image intensity gradient.

$$\nabla G(x,y) * I(x,y), \qquad where \;\; \nabla = (\partial/\partial x, \partial/\partial y) \tag{3.2}$$

and

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-x_0)^2+(y-y0)^2}{2\sigma^2}} \tag{3.3}$$

The voting procedure is carried out using Hough Transforms [16, 17, 22] on the parametric definitions of the iris boundary contours.
Hough transform is given in the following equation :

$$H(x_c, y_c, r) = \sum_{j=1}^{n} h(x_j, y_j, x_c, y_c, r) \tag{3.4}$$

where

$$h(x_j, y_j, x_c, y_c, r) = \begin{cases} 1 & \text{if } g(x_j, y_j, x_c, y_c, r) = 0 \\ 0 & \text{otherwise} \end{cases}$$

with

$$g(x_j, y_j, x_c, y_c, r) = (x_j - x_c)^2 + (y_j - y_c)^2 - r^2$$

3.2.3 Other Schemes

Ritter et.al [27] uses active contour models for localization of the pupil. Some authors have also worked to further improve the performance of localization by the detection of eyelids and eyelashes [25] to further improve the time of localization. The preprocessing phase which deals with the localization and normalization of Iris image considers the pupil and iris boundaries to be circular in shape. Considerable work has also been done for the detection of eyelids and eyelashes.

A technique proposed by Huang et. al [14] tries to improve the iris localization time by detecting the outer iris boundary in the rescaled image. Scaling the image for an optimum fraction of the original performs much better than the original image. Many other authors have also proposed methods to detect the eyelids and eyelashes. The detection of eyelids involves the search for two curves, which satisfy the polynomial equation of the form :

$$x(t) = at^2 + bt + c, \quad t \in [0, 1]. \tag{3.5}$$

Eyelashes can be detected by checking the variance for each block. Some authors have also used threshold based approaches to find the coarse localization of pupil. Since the pupil is the darkest region, a certain threshold over the intensity separates the pupil boundary.

Kong and Zhang [11] proposed a method for the detection of eyelashes. He

divided the eyelashes into two types. First one is separable eyelashes which can be easily separated from the image and the second one being multiple eyelashes, which are bunched together and are found overlapping in the eye image. One dimensional Gabor filters can be used to detect separable eyelashes because the gaussian smoothing of a separable eyelash results into a low output value. Thus the resultant points smaller than a threshold represent points on the eyelash. To detect multiple eyelashes, the variation of intensity method is used. If the variance of the intensity values in a window is lower than a threshold value than the center of the window is a point on the eyelash.

3.3 Parallel Approach to Iris Recognition

Works in parallel iris recognition started quite late. This got a boost with the development of parallel computing infrastructure like multicore architecture and new technologies for the development of parallel programs. Anyhow, Daugman himself suggested the introduction of parallelism in the matching step of the IrisCode generated from the image against the template IrisCodes present in the database. The matching process involves the comparison of IrisCodes by carrying out Boolean XOR operations to compute a function called Hamming distance, which finds out the amount of dissimilarity between any two iris codes [4, 6].

$$HammingDistance = \frac{||(CodeA \oplus CodeB) \cap MaskA \cap MaskB||}{||MaskA \cap MaskB||} \tag{3.6}$$

The Hamming distance can be computed very quickly given the manner a computer handles the bitwise operations. Bitwise operations are done directly at the hardware level and they are very efficient. It takes only 1.7 seconds to compare the IrisCodes of a million templates on a 2.2 GHz computer. This level of performance enables the iris recognition system to be applied for large scale applications where the calculated IrisCode has to be compared against millions of templates. Since the

IrisCode is of size 2048 bits, the comparisons could be done in parallel in terms of 32 bit chunks.

There have been some notable works recently in applying parallel approach towards solving the problem of iris recognition. In 2011 Fatma Zaky Sakr et. al. [18, 19] presented an alternative method by the use of direct parallel processing, using the Graphics Processing Units. In their work titled 'High Performance Iris Recognition System on GPUs', they have parallelized iris recognition after the deconstruction of the most time consuming steps involved in the process of Iris Recognition. In particular they have focused on the matching and identification stages since they require the maximum amount of computation if the size of the database to compare is extremely high. They have applied Daugman's algorithm in every step of the process. The speedup obtained was around 10 to 15 times in the matching and identification phases and around 1.3 times when taking all the stages of iris recognition into consideration. They implemented the system on an NVIDIA GTX 460 GPU with 336 cores.

The identification process according to Daugman's methodology consists of

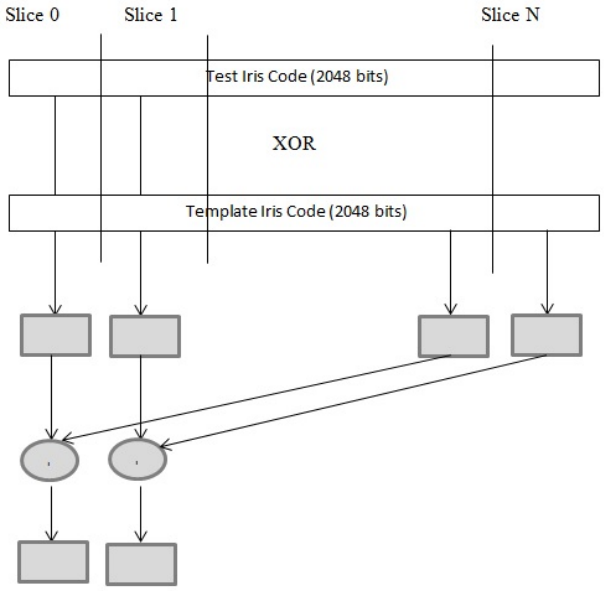

Figure 3.2: Parallel Computation of the Hamming Distance on the GPU

matching the 2048 bit IrisCodes for the images by calculating the Hamming distance between the input code and the template codes. [18] explores this fact by matching the codes in parallel. The code is divided into bit slices and a single bit slice of 32 bits is compared on a single processor by XORing the input code with the template codes as shown in figure 3.2.

In 2012, they came up with another work titled 'Accelerating Iris Recognition Algorithm on GPUs' in which they focused on localization and extraction processes, since these are the two most time consuming steps compared to the matching step. They again implemented an Iris recognition system based on Daugman's system on an NVIDIA GTX 460 GPU with 336 cores. They obtained a speedup of 9.6 and 12.4 for the two processes and a total speedup of 12.4 considering all the processes.

In another approach, Ryan N. Rakvik et al. [20, 21] present a parallel processing alternative using Field Programmable Gate Arrays (FPGAs), which accelerates the application to a much better level. They have used parallel processing to speed up most time consuming steps involved in the process of iris recognition. Iris Localization, template generation and matching steps require a lot of computing steps and its parallelization has been done on an FPGA system, providing an average speedup of 9.6 for the localization step, 324 for template generation step and 19 for matching step compared to a CPU based system. FPGAs are programmable logic devices with integrated circuits that can be programmed as per requirement.

3.4 Summary

A lot of work has been done in the field of iris recognition but most of the work has been done keeping the sequential CPU based systems in mind. To summarize, we see that a lot of scope still remains in the full utilization of the GPU for the purpose of localization and feature extraction which are the most computation intensive parts of Iris recognition process.

Chapter 4

Parallel Iris Localization

Objective

Most of the Iris Localization techniques developed so far are sequential in nature. The objective of the thesis is to implement an existing Iris Localization technique over a parallel system to speedup the task. The accuracy of the algorithm is not of concern since it has already been tested by their developers. For the purpose of Iris localization, Hough transform method has been chosen for the detection of circular boundaries, and, GPU with CUDA architecture has been chosen as the parallel system for implementation. The reason for using Hough transform is that, it is a robust and effective method while dealing with noisy and incomplete data, unlike several threshold based approaches.

4.1 Traditional Hough Transform

Traditional Hough Transform is a very effective technique when it comes to the identification of basic shapes present in an image. Initially the Hough transform was used only to find the presence of lines in an image but subsequently it was extended for the identification of other shapes like circles and ellipses. The concept of Hough Transform was put forward by Richard Duda and Peter Hart [16] in 1972, and they termed it as Generalized Hough Transform.

The output of an edge detection algorithm is fed as the input to the Hough transformation step. The edge detection algorithm detects the sharp edges of various shapes present in the image by the application of gradient checks on the image. Subsequent algorithm then detects a particular shape present in the edge image. Hough transform is very effective for noisy data where there may be several imperfections in the image as well as in the edge detection algorithm, but it comes with a cost of computational complexity which is quite large.
Hough transform is given in the following equation :

$$H(x_c, y_c, r) = \sum_{j=1}^{n} h(x_j, y_j, x_c, y_c, r) \tag{4.1}$$

where

$$h(x_j, y_j, x_c, y_c, r) = \begin{cases} 1 & \text{if } g(x_j, y_j, x_c, y_c, r) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

with

$$g(x_j, y_j, x_c, y_c, r) = (x_j - x_c)^2 + (y_j - y_c)^2 - r^2 \tag{4.3}$$

Various improvements have been made over the traditional Hough transform [15].

Some authors have used Canny edge detectors with Hough transform for an improvement in the speed of the localization [16]. Some authors have used threshold based techniques for the identification of the pupil but it doesn't always give desired result, given that the color of the iris may vary a lot ranging from light colored to dark colored. The difficulty increases for more dark irises.

Hough Transform method groups the edge points into recognizable shapes like a line, circle or an ellipse. Since the iris and pupil are circular in shape, Hough Transform for circles [16, 17] is used for the detection of pupil and iris boundaries. A circle is described completely through three variables, i.e. its center (a, b) and radius r.
The parametric equation of a circle is :

$$x = a + rcos\theta \tag{4.4}$$

$$y = b + rsin\theta \tag{4.5}$$

Varying the value of θ from 0 to 360 degrees yields the complete circle. In the circle Hough transform we expect to find a triplet (x, y, r) that is highly probable to be a circle in the image. Assuming that we are looking for circles of a particular radius, i.e. the radius is known. Every point in the image space will be equivalent to a circle in the Hough space, i.e. Parameter space.
Rearranging the above equations we get:

$$a = x - rcos\theta \tag{4.6}$$

$$b = y - rsin\theta \tag{4.7}$$

These are the equations of the circle in the parameter space corresponding to the particular point (x, y) in the image space.

The serial Hough transform starts with the output of an edge detector algorithm as its input. Then the maximum and the minimum radius r_{max} and r_{min} are defined over which to search for a circle. For each possible radius r where $r_{min} \leq r \leq r_{max}$ an accumulator array $houghspace_i$ is created with a size equal to the number of quantized pixels in the image, i.e.$width * height$ of the image. The accumulator is initialized to trivial value 0. All possible circles with the particular radius value r are drawn in the parameter space. Votes are added to the accumulator cells for all the pixels which lie on the circle drawn in the parameter space, if the center of the drawn circle is an edge point in the image space. A high value in the accumulator represents that the pixel falls on the intersection point of many circles in the parameter space. This intersection point represents a potential point for the center of a circle in the image space.

Since the radius of the circle to be searched is actually not known in advance, a simple solution is to guess for circles of all probable radii with the maximum radius equal to the length of the diagonal of the image, because no circle will have a radius of length greater than the length of the diagonal of the image. A search for the circles with all possible radii is done, and accumulators are maintained for all the searches, i.e. One accumulator array for the search of a circle with a particular radius. The highest values from each accumulator cell are gathered in another list, and the highest values in this list represent the potential circles. The detailed steps followed in the process is shown in Algorithm 1.

4.2 Parallel Algorithm

Algorithm 2 describes the detailed steps followed while mapping the algorithm on GPU. The initial steps are same as the serial algorithm. For each possible radius value, a CUDA grid of size $(width - 2 * r) * (height - 2 * r)$ blocks is created. The size of each block is 72 threads. Each thread calculates the position for a pixel in the hough space, then loads a image pixel from the circle edge image and then updates the accumulator for the particular radius.

4.3 Time Complexity Analysis

Let the size of the image be $imgsize = width * height$ pixels. For the serial algorithm, in the extreme case the minimum radius could be 1 pixel and the maximum radius could be equal to the length of the diagonal of the image, $d = \sqrt{width^2 + height^2}$ pixels. So the total number of possible radius values is equal to the diagonal of the image d.

For each radius value $radius$, we create approximately $width * height$ number of circles in the hough space, assuming that the size of the image is comparatively large compared to the radius values. For each circle we calculate 72 points.

Algorithm 1 SerialLocalize ($Image$)

1: $img \leftarrow LoadImage(Image)$
2: Apply edge detector to produce an edge image $\acute{img}$.
3: Define array $imgValue$ with value 1 for all edge pixels and 0 for others.
4: Define minimum and maximum search radius as r_{min} and r_{max}
5: Create arrays $maxValues$, $xPositions$, $yPositions$ and $radiusValues$ all of size $r_{max} - r_{min}$
6: $m \leftarrow 72$
7: $i \leftarrow 0$
8: **for** $r \leftarrow r_{min}$ *to* r_{max} **do**
9: $n \leftarrow (width - 2 * r) * (height - 2 * r)$
10: Create accumulator array $houghspace_i$ of size n and initialize with all 0.
11: **for** Circle $C_1, C_2, ..., C_n$ each of radius value r **do**
12: **for** Points $P_1, P_2, ..., P_m$ on circle C_j **do**
13: **if** Center of the circle on which P_k lies, represents an edge point **then**
14: Increment $houghspace_i$ at the edge point.
15: **end if**
16: **end for**
17: **end for**
18: Find $max(houghspace_i)$ and add this value to $maxValues[i]$ and update other arrays $radiusValues$, $xpositions$ and $yPositions$ according to this value.
19: $i \leftarrow i + 1$
20: **end for**
21: $max(maxValues)$ corresponds to a circle in image space.

Algorithm 2 ParallelLocalize ($Image$)

1: $img \leftarrow LoadImage(Image)$
2: Apply edge detector to produce an edge image $\acute{img}$.
3: Define array $imgData$ with value 1 for all edge pixels and 0 for others.
4: Define minimum and maximum search radius as rad_{min} and r_{max}
5: Create arrays $maxValues$, $xPositions$, $yPositions$ and $radiusValues$ all of size $r_{max} - r_{min}$
6: $i \leftarrow 0$
7: **for** $r \leftarrow r_{min}$ *to* r_{max} **do**
8: Create accumulator $houghspace_i$ and initialize with all 0.
9: $gridSize = (width - 2 * r) * (height - 2 * r)$ blocks
10: $blockSize \leftarrow 72 threads$
11: Call **parallel** $Kernel(img, r, houghspace_i)$ for each thread
12: Find $max(houghspace_i)$ and add this value to $maxValues[i]$.
13: Record corresponding radius and center
14: Update other arrays $radiusValues$, $xPositions$ and $yPositions$ accordingly.
15: $i \leftarrow i + 1$
16: **end for**
17: $max(maxValues)$ corresponds to a circle in image space .

Algorithm 3 Kernel ($img, radius, houghspace$)

1: Calculate position of a pixel drawn on a circle in the hough space
2: Load the image pixel at the corresponding center position pos from the circle edge image.
3: Get the pixel value $pixValue$ at the position pos
4: **if** $pixValue > 0$ **then**
5: Update $houghspace$ at the pixel value pix in the hough space
6: **end if**

The time complexity of the serial algorithm is proportional to $d * imgsize * 72$.

$$T_s = O(d * imgsize) \tag{4.8}$$

For the parallel algorithm also we deal with d number of radius values. For each radius value we create $width * height$ number of blocks, each with 72 threads. Each block evaluates a single circle in the hough space.

Since $width * height * 72$ number of threads run in parallel, the time complexity is proportional to the following:

$$T_p = O(d) \tag{4.9}$$

The above equation holds if there are sufficient number of cores to handle each thread independently. For lesser number of cores the time complexity will be a multiple of T_p.

Chapter 5

Implementation and Results

The sequential CPU implementation serves as the basis for the implementation of the parallel algorithm and the comparison of its performance against the serial one. The CPU implementation is quite straightforward in accordance to the algorithm discussed before. The basic idea behind the parallel implementation of Iris Localization algorithm is the fact that the data can be decomposed into smaller parts that can be executed in parallel.

5.1 Implementation Environment

For the purpose of implementation of the parallel algorithm, a low end graphics processing unit has been chosen which still provides wonderful results to the speedup of the application compared to a general purpose CPU based system.

The total number of CUDA cores plays a significant role in the execution time of the algorithm. Higher number of cores are poised to give better results, of course with some limitations. Also the bandwidth of the memory interface between the GPU cores and the GPU memory plays an important part.

The details of the environment and hardware used for the application is shown in the table 5.1.

Item	**Details**
GPU	NVIDIA GeForce 410M GPU
GPU Specifications	48 CUDA cores, 512 MB GPU Memory
Processor	Intel Core i3 2330M CPU, 2.20 GHz
System Memory	2 GB
Operating System	Windows 8

Table 5.1: Details of the implementation environment

Sample No.	$t_{sequential}$**(sec)**	$t_{parallel}$**(sec)**	**Speedup**
1	7.424	1.362	5.45
2	7.391	1.344	5.50
3	7.256	1.354	5.36
4	7.364	1.299	5.67
5	7.448	1.350	5.52
6	7.420	1.341	5.53
7	7.334	1.339	5.48
8	7.329	1.352	5.42
9	7.452	1.358	5.49
10	7.399	1.365	5.42

Table 5.2: Serial and parallel execution times of the localization algorithm on sample random images of size 320x280 pixels from the CASIA database

5.2 Results

Table 5.2 shows the algorithm execution time for the detection of the circular portions i.e. pupil and iris in the random eye images obtained from the standard CASIA database. The average speedup obtained on the GPU is **5.48x** for sample from the CASIA database [28]. Though the number of processor cores in the GPU used is 48, the Speedup is not 48 because of a multitude of factors like amount of sequential code, and communication latency as discussed in Chapter 1. This is a considerable improvement, given the fact that the actual speedup will be much

Sl No.	**Image Size(px)**	$t_{sequential}$ (sec)	$t_{parallel}$ (sec)	Speedup
1	64x56	0.078	0.031	2.51
2	128x112	0.593	0.125	4.74
3	192x168	1.672	0.344	4.86
4	256x224	3.844	0.735	5.23
5	320x280	7.424	1.362	5.45
6	384x336	12.469	2.219	5.62
7	448x392	19.939	3.437	5.80
8	512x448	29.996	5.801	6.00

Table 5.3: Execution times of the Iris localization algorithm on iris images of different sizes

better on the actual dedicated systems for the purpose of iris recognition, since the implementation environment chosen for the implementation is affected by a multitude of factors like hundreds of processes executing on the processor at any given time, and many other factors.

Another important result obtained from the experiment was the speedup obtained for different sizes of the images. The results show that for images of different sizes, obtained by scaling original images from the CASIA database, the speedup increases exponentially. This shows that for less computation intensive task, i.e. while dealing with smaller images, the GPU hardware remains underutilized. The values are tabulated in table 5.2. The table shows the execution times on the CPU and GPU for images of different sizes.

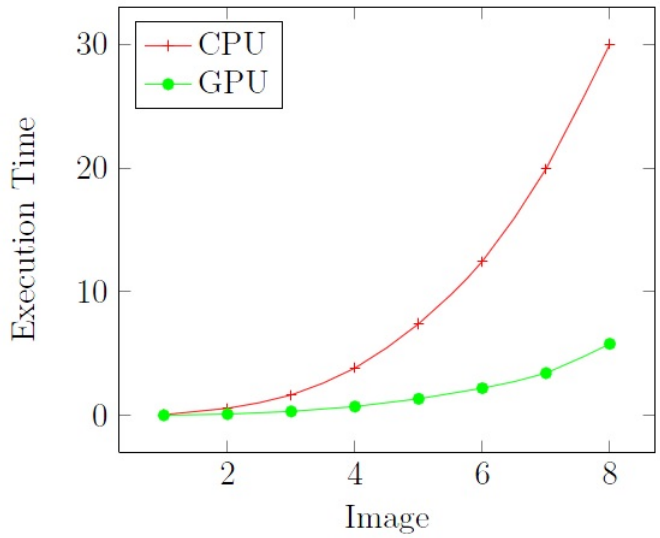

Figure 5.1: Execution Time vs Image Size. The image size is obtained using Image Serial No. from table 5.3

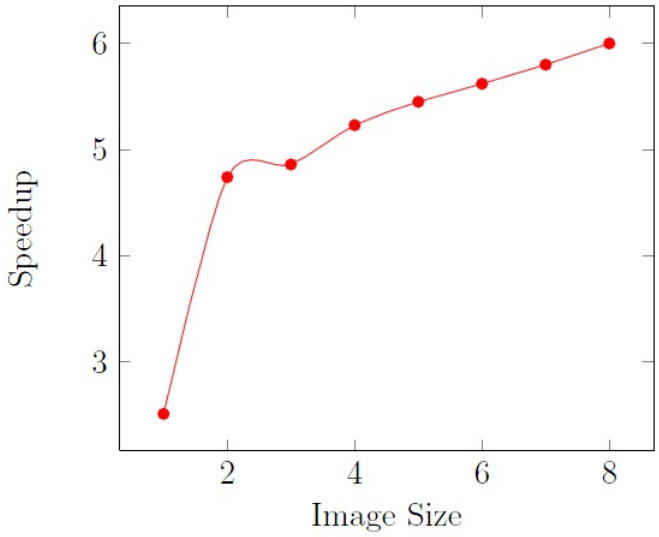

Figure 5.2: Variation of speedup with respect to image size. Image size is obtained using Image Serial No, from table 5.3

Chapter 6

Conclusion

In this thesis, a vital part of Iris recognition, i.e. Iris localization is parallelized on Graphics Processing Unit(GPU), using NVIDIA's CUDA architecture. CUDA allows us to express a complex parallel algorithm in a standard high level language like C. The flexibility and expressiveness of CUDA facilitates the parallel implementation of Iris localization, that can be included in a parallel iris recognition system.

In addition this thesis also shows the underutilization of the GPU if not exploited properly. The increase in speedup versus the image size shows this fact. Hence in the future, care has to be taken to fully utilize the GPU, if possible. Also, if all the steps of Iris recognition are accelerated using GPUs, the performance will be greatly increased, and the use it can find will be enormous. With the recent projects such as UIDAI where images of millions of people are enrolled, the benefit will be even more, while searching for a record in the database.

GPUs have revolutionized the way computing is done, and it will continue to happen so in the future with more and more parallel versions of existing sequential algorithms being created for execution on the GPUs. There i no doubt that image processing applications will turn out to be among the best gainers of this revolution.

Bibliography

[1] Ananth Grama, Anshul Gupta, George Karypis, and Vipin. *Introduction to Parallel Computing*. Pearson Education, Second Edition, 2007.

[2] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programming*. Addison-Wesley Professional, First Edition, 2004.

[3] L. Flom and A. Safir. *Iris Recognition Systems*. US Patent 4641349, 1987.

[4] J. G. Daugman: *High confidence visual recognition of persons by a test of statistical independence*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol 15, pp.1148 - 1161, 1993.

[5] J. G. Daugman. *Biometric personal identification system based on iris analysis*. US Patent 5291560, 1994.

[6] J. G. Daugman. *How Iris Recognition Works*. IEEE Transactions on Circuits and Systems for Video Technology, Vol 14(1), pp. 21 - 30, 2004.

[7] J. G. Daugman. *The importance of being random : statistical principles of iris recognition*. Pattern Recognition, Vol 36(2), pp. 279 - 291, 2003.

[8] R. Wildes, J. Asmuth, G. Green, S. Hsu, R. Kolczynski, J. Matey, S. McBride: *A Machine-vision System for Iris Recognition*. Machine Vision and Applications, Vol 9, pp. 1-8

[9] D.G. Lowe. *Distinctive image features from scale invariant keypoints*. International Journal of Computer Vision, Vol 60(2), pp. 91-110, 2004.

[10] J. Huang, Y. Wang, T. Tan, and J. Cui, *A new iris segmentation method for recognition*, 17th International Conference on Pattern Recognition, vol. 3, pp. 554557, August 2004.

[11] W. Kong and D. Zhang, *Accurate iris segmentation based on novel reflection and eyelash detection model*. International Symposium on Intelligent Multimedia, Video and Speech Processing, pp. 263266, May 2001.

[12] Q. Tian, Q. Pan, Y. Cheng, and Q. Gao, *Fast algorithm and application of Hough transform in iris segmentation.* International Conference on Machine Learning and Cybernetics, Vol. 7, pp. 39773980, August 2004.

[13] A. Bachoo and J. Tapamo, *A segmentation method to improve iris based person identification*, in proceedings of 7th Africon Conference, vol. 1, pp. 403408, September 2004.

[14] Y. Huang, S. Luo, and E. Chen. *An efficient iris recognition system.* International Conference on Machine Learning and Cybernetics, volume 1, pages 450454, 2002.

[15] Y. Liu, S. Yuan, X. Zhu, and Q. Cui. *A practical iris acquisition system and a fast edges locating algorithm in iris recognition.* 20th IEEE Conference on Instrumentation and Measurement Technology, volume 1, pages 166168, 2003.

[16] R.O.Duda, P.E.Hart. *Use of the Hough Transformation to Detect Lines and Curves in Pictures.* Graphics and Image processing, Communications of the ACM. Vol 15, No. 1, January 1972.

[17] P.Hough. *Method and Means for Recognizing Complex Patterns.* US Patent No. 3069654, 1962

[18] F.Z. Sakr, M. Taher, A.M. Wahba. *High Performance Iris Recognition System On GPU*, IEEE International Conference on Computer Engineering & Systems (ICCES), pp. 237 242, 2011.

[19] F.Z. Sakr, M. Taher, A.M. Wahba. *Accelerating Iris Recognition algorithms on GPUs*, IEEE CIBEC, Cairo, Egypt, 2012.

[20] Ryan N. Rakvic, Bradley J. Ulis, Randy P. Broussard, Robert W. Ives and Neil Steiner. *Parallelizing Iris Recognition.* IEEE transactions on Information Forensics and Security, Vol. 4, No 4, December 2009.

[21] R. N. Rakvic, B. J. Ulis, R. P. Broussard, and R. W. Ives. *Iris template generation with parallel logic*, in Proceeding 42nd Annual Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, Nov. 2008.

[22] J. Illingworth and J. Kittler, *A survey of the Hough transform,* Computer Vision, Graphics and Image Processing, vol. 44, pp. 87116, 1988.

[23] NVIDIA Corporation, *CUDA C Programming Guide*, www.developer.nvidia.com/cuda, Version 3.1, 2010.

[24] NVIDIA Corporation, *CUDA C Reference Manual*, www.developer.nvidia.com/cuda, Version 3.1, 2010.

[25] K.W. Bowyer, K. Hollingsworth, and P. J. Flynn. *Image understanding for iris biometrics: A survey.* Computer Vision and Image Understanding, Vol. 110(2), pp. 281307, 2008.

[26] R.P. Wildes *Iris recognition: an emerging biometric technology.* Proceedings of the IEEE, Vol. 85, No. 9, pp. 13481363, 1997.

[27] N. Ritter. *Location of the pupil-iris border in slit-lamp images of the cornea.* International Conference on Image Analysis and Processing, 1999.

[28] Chinese Academy of Sciences Institute of Automation. Database of 756 Greyscale Eye Images. www.sinobiometrics.com