

Learning Resource Center

BOOK NUM.:

This book was issued is overdue due on the date stamped below. If the book is kept over due, a fine will be charged as per the library rules.

[illegible]

PARALLEL IRIS RECOGNITION

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

By

ANKIT ANAND-081205

KARTIK GODAWAT-081314

Under the supervision of

Mr. RAVIKANT VERMA & Mrs. MEENAKSHI ARYA



MAY – 2012

Department of Computer Science Engineering

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,

WAKNAGHAT

Contents

S. No	Chapter No.	Topic	Page No.
1		Certificate	i
2		Acknowledgement	ii
3		List of Figures	iii
4		Summary	iv
5	1	Introduction	1
6	2	CUDA Architecture	3
7	3	Program Structure	6
8	4	Hardware & Tools	8
9	5	Initial Work	9
10	6	Iris Recognition	12
11	7	Results & conclusion	18
12	8	Future Work	22
13		References	23
14		Code	24

CERTIFICATE

This is to certify that the work titled "**Parallel Iris Recognition**" submitted by "**Ankit Anand & Kartik Godawat**" in partial fulfillment for the award of degree of Bachelor of Technology of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.



Mr. Ravikant Verma

Senior Lecturer, JUIT

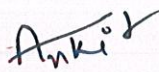
Date : 31. May '12

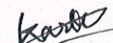
Acknowledgement

First of all, with profound veneration and humble submission, we would like to express our sense of gratitude to Brig (Retd.) S. P. Ghrera, HOD (CSE department), Jaypee University of Information Technology, Solan for giving us the opportunity to work on this project and providing us valuable guidance.

We would also like to express our appreciation to our project guides Mr. Ravikant Verma & Mrs Meenakshi Arya. We wholeheartedly thank them for their support and help. We feel motivated and encouraged everytime we discuss our tribulations to them. Without their encouragement and guidance this project would not have been possible

Date: 31 MAY, 2012


Ankit Anand (081205)


Kartik Godawat (081314)

List of figures

S No.	Figure No.	Title	Page No.
1	2.1	Architecture of a CUDA Capable GPU	3
2	2.2	The Fermi Architecture	5
3	3.1	Execution of a typical CUDA program	6
4	3.2	Processing Flow of CUDA	7
5	5.1	Negative Of image	10
6	5.2	Parallel Rank Sort	11
7	6.1	Test Image	16
8	6.2	Image Centered about the mean	16
9	6.3	Iris Recognition Design flow	17
10	7.1	Image Negative Snapshot	18
11	7.2	Rank Sort Snapshot	18
12	7.3	Iris Recognition Input Screen	19
13	7.4	Iris Recognition output Screen	19
14	7.5	CPU time vs. GPU time	20
15	7.6	Observed Speed Up	21

Summary

CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). Traditionally parallel processing implied setting up a Distributed system but, with the advent of CUDA the power of parallel processing can be harnessed by a single standalone machine. Anyone with a basic knowledge of C/C++ and good programming skills can exploit CUDA.

The biometrics trait iris - the colored circle that surrounds the pupil - contains many randomly distributed immutable structures, which makes each iris distinct from another. Like fingerprint, iris does not change with time. Iris recognition is a relatively new biometric recognition technique. Among the various traits, iris recognition has attracted a lot of attention because it has various advantageous factors like simplicity and accuracy compared to other biometric traits. Iris recognition relies on the unique patterns of the human iris to identify or verify the identity of an individual. It uses camera technology with subtle infrared illumination to acquire images of the detail-rich, intricate structures of the iris.

However, due to the large databases that the governments around the world intent to keep the biometric recognition techniques are not sufficiently fast. Analyzing the data of millions of people could take hours or may be days. Hence, we have created a system that speeds up this process by utilizing the immense potential of parallel processing using CUDA.

Chapter 1: Introduction

With today's fast growing computational requirements, the current hardware has reached the limits of integration capabilities on a single chip using known technologies. If it is increased any further increase it will put constraints of working conditions on the chips.

As a result, since 2003, the semiconductor industry has settled on two main directions for designing microprocessor. The multicore microprocessor aims at maintain the execution speed of sequential programs while moving into multiple cores. The multicores began as two-core processors, with the number of cores approximately doubling with each semiconductor process generation. A current example is the recent Intel Core i7 microprocessor, which has four processor cores, each of which is an out-of-order, multiple-instruction issue processor implementing the full x86 instruction set; the microprocessor supports hyperthreading with two hardware threads and is designed to maximize the execution speed of sequential programs.

However, the many-core microprocessors focus more on the execution throughput of parallel applications. The many-cores started as a large number of much smaller cores, and, once again, the number of cores doubles with each generation. A current example is the NVIDIA GeForce GTX 280 graphics processing unit (GPU) with 240 cores, each of which is a heavily multithreaded, in-order, single-instruction issue processor that shares its control and instruction cache with seven other cores. Many-core processors, especially the GPUs, have led the race of floating-point performance since 2003. While the performance improvement of general-purpose microprocessors has slowed significantly, the GPUs have continued to improve relentlessly. As of 2009, the ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 10 to 1. These are not necessarily achievable application speeds but are merely the raw speed that the execution resources can potentially support in these chips: 1 teraflops (1000 gigaflops) versus 100 gigaflops in 2009.

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia for graphics processing. CUDA is the computing engine in Nvidia GPUs (Graphics Processing Units) which is accessible to software developers through a variety of industry standard programming languages. Programmers can use 'C for CUDA' which is C with Nvidia extensions and certain restrictions. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Using CUDA, the latest

Nvidia GPUs become accessible for computation like CPUs. Unlike CPUs however, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general purpose problems on GPUs is known as GPGPU.

CUDA provides both a low level API and a higher level API. The initial CUDA SDK was made public on 15 February 2007, for Microsoft Windows and Linux. Mac OS X support was later added in version 2.0, which supersedes the beta released February 14, 2008. CUDA works with all Nvidia GPUs from the G8x series onwards, including GeForce, Quadro and the Tesla line. CUDA is compatible with most standard operating systems. Nvidia states that programs developed for the G8x series will also work without modification on all future Nvidia video cards, due to binary compatibility.

Chapter 2: The CUDA Architecture

Figure 2.1 shows the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Figure 2.1, two SMs form a building block; however, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. Also, each SM in Figure 2.1 has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with up to 4 gigabytes of graphics double data rate (GDDR) DRAM, referred to as global memory in Figure 2.1. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold video images, and texture information for three-dimensional (3D) rendering, but for computing they function as very-high-bandwidth, off-chip memory, though with somewhat more latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

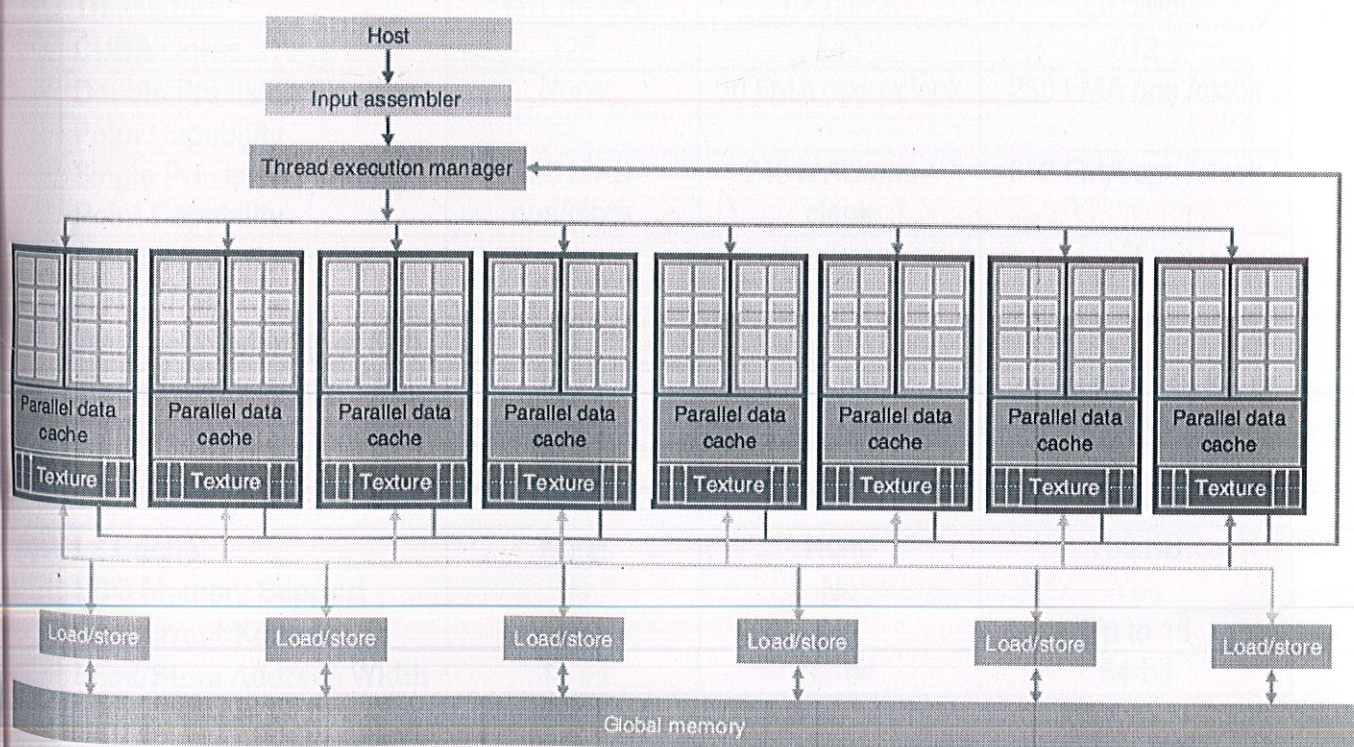


Figure 2.1: Architecture of a CUDA capable GPU

The G80 that introduced the CUDA architecture had 86.4 GB/s of memory bandwidth, plus an 8-GB/s communication bandwidth with the CPU. A CUDA application can transfer data from the system memory at 4 GB/s and at the same time upload data back to the system memory at 4 GB/s. Altogether, there is a combined total of 8 GB/s. The massively parallel G80 chip has 128 SPs (16 SMs, each with 8 SPs). Each SP has a multiply-add (MAD) unit and an additional multiply unit. With 128 SPs, that's a total of over 500 gigaflops. In addition, special function units perform floating-point functions such as square root (SQRT), as well as transcendental functions. With 240 SPs, the GT200 exceeds 1 teraflops. Because each SP is massively threaded, it can run thousands of threads per application. A good application typically runs 5000–12,000 threads simultaneously on this chip. For those who are used to simultaneous multithreading, note that Intel CPUs support 2 or 4 threads, depending on the machine model, per core. The G80 chip supports up to 768 threads per SM, which sums up to about 12,000 threads for this chip. The GT200 supports 1024 threads per SM and up to about 30,000 threads for the chip. The latest architecture from Nvidia, the Fermi architecture, supports billions of threads.

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Source: Fermi Whitepaper

Table 2.1 A comparison between G80, GT200 & Fermi architectures

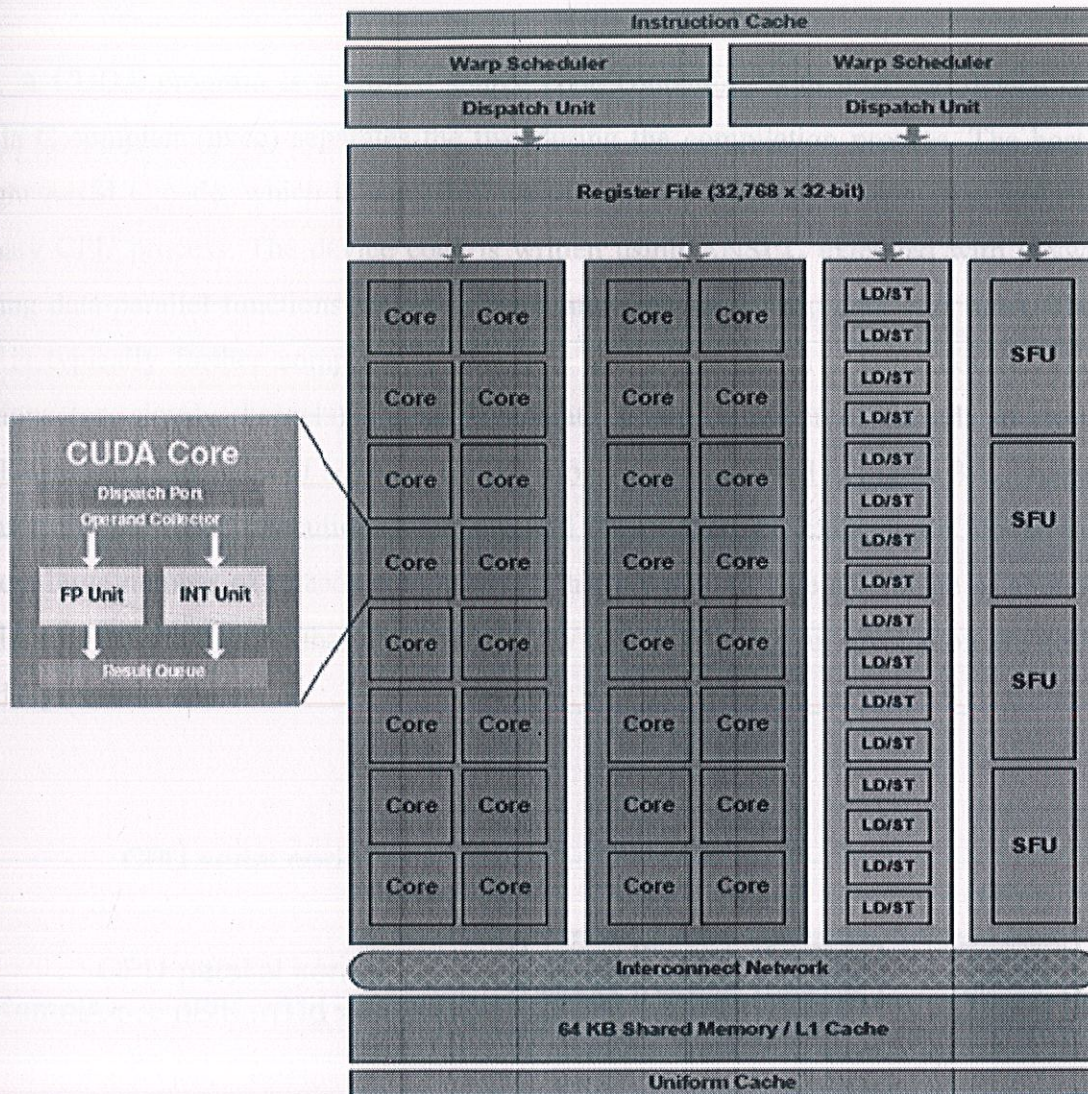
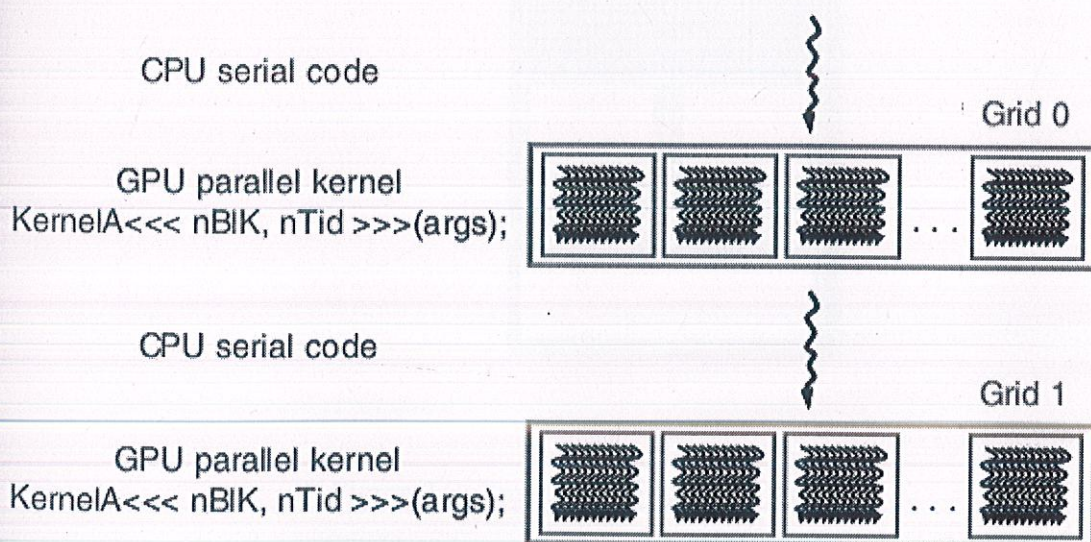


Figure 2.2: The Fermi Architecture

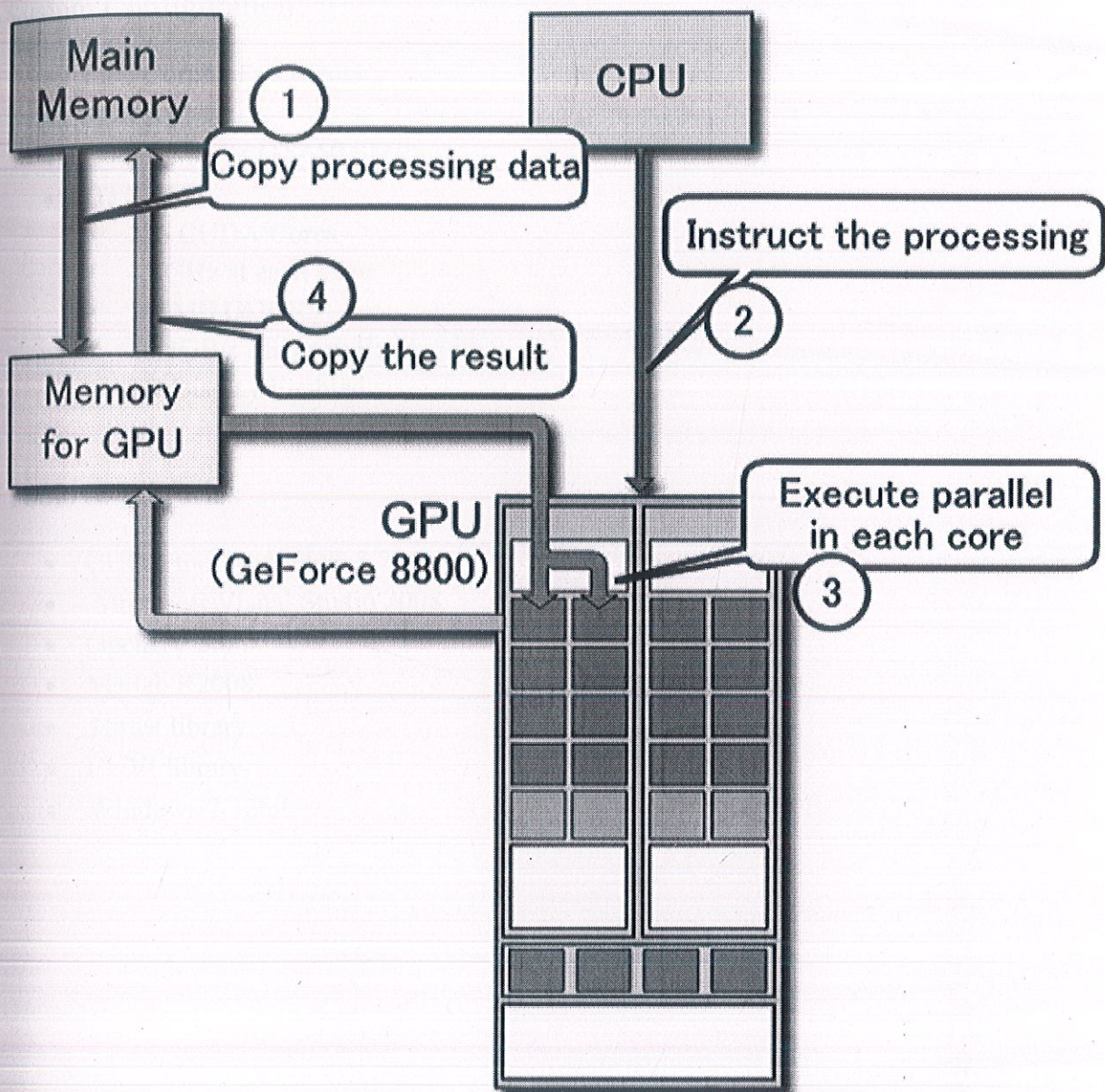
Chapter 3: The CUDA program Structure

A CUDA program consists of one or more parts that are executed on either the CPU (i.e. the Host) or a GPU (i.e. a device). The parts that show little or no data parallelism are implemented in host code. The parts that show rich amount of data parallelism are implemented in the device code. A CUDA program is a unified source code containing both host and device code. The Nvidia C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code, which is compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures. The device code is typically further compiled by the nvcc and executed on a GPU device. The kernel functions (or, simply, kernels) typically generate a large number of threads to exploit data parallelism. The execution of a typical CUDA program is illustrated in Figure 3.1. The execution begins with CPU (host) execution. When a kernel `f` is invoked, the execution is moved to a GPU, where a large number of threads are generated that are able to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a grid.



Source: Programming massively parallel processors

Figure 3.1: Execution of a typical CUDA program



Source: www.wikipedia.com

Figure 3.2: Processing flow of CUDA

Chapter 4: Hardware & Tools

System Configuration

- Intel Core i3 Processors
- 4GB RAM
- Cooler Master GX650 SMPS
- GTX 460
 - 336 CUDA Cores
 - 1.3GHz at each Core
 - 768MB GDDR5
 - 86.4GB/s Memory Bandwidth
 - 3D Display Enabled

Tools & Libraries

- CUDA toolkit & SDK 3.2
- .Microsoft Visual Studio 2008
- OpenCV 3.0
- Matlab R2008
- Thrust library
- CUVI library
- Windows 7 32bit

Chapter 5: Initial Work

Negative of an image

Images have a finite set of digital values, called picture elements or pixels. A digital image consists of a fixed number of rows and columns of pixels. Pixels are the smallest individual element in an image, holding quantized values that represent the brightness of a given color at any specific point.

Typically, the pixels are stored in computer memory as a raster image or raster map, a two-dimensional array of small integers. These values are often transmitted or stored in a compressed form.

Raster images can be created by a variety of input devices and techniques, such as digital cameras, scanners, coordinate-measuring machines, seismographic profiling, airborne radar, and more. They can also be synthesized from arbitrary non-image data, such as mathematical functions or three-dimensional geometric models; the latter being a major sub-area of computer graphics. The field of digital image processing is the study of algorithms for their transformation.

The number of distinct colors that can be represented by a pixel depends on the number of bits per pixel (bpp). A 1 bits per pixel image uses 1-bit for each pixel, so each pixel can be either on or off, which is mostly the case with black & white images. Each additional bit doubles the number of colors available, so a 2 bpp image can have 4 colors, and a 3 bpp image can have 8 colors:

- 1 bpp, $2^1 = 2$ colors (monochrome)
- 2 bpp, $2^2 = 4$ colors
- 3 bpp, $2^3 = 8$ colors
- 8 bpp, $2^8 = 256$ colors
- 16 bpp, $2^{16} = 65,536$ colors ("Highcolor")
- 24 bpp, $2^{24} \approx 16.8$ million colors ("Truecolor")

For color depths of 15 or more bits per pixel, the depth is normally represents the sum of the bits allocated to each of the three components red, green, and blue. The highcolor, which usually means 16 bpp, normally has five bits for red and blue, and six bits for

green, as the human eye is more sensitive to errors in green than in the other two primary colors. For applications involving transparency, the 16 bits may be divided into five bits each of red, green, and blue, with one bit left for transparency. A 24-bit depth allows 8 bits per component. On some systems, 32-bit depth is available: this means that each 24-bit pixel has an extra 8 bits to describe its opacity (for purposes of combining with another image).

Portable Network Graphics is a bitmapped image format that employs lossless data compression. PNG was created to improve upon and replace GIF (Graphics Interchange Format) as an image-file format not requiring a patent license. PNG supports palette-based images (with palettes of 24-bit RGB or 32-bit RGBA colors), grayscale images (with or without alpha channel), and full-color non-palette-based RGB[A] images (with or without alpha channel). PNG was designed for transferring images on the Internet, not for professional-quality print graphics, and therefore does not support non-RGB color spaces such as CMYK.

The negative of an image is computed by inverting each of its pixel values (RGB). Inversion means subtracting each of the Components of each pixel from 255



Figure 5.1: Negative of image

Rank Sort

For each element of the list to be sorted, the Rank Sort algorithm is computing the total number of elements that are lower than that number. This value is called the rank of the element and to compute it the algorithm needs to compare the element with all other values from the list. In a fully sorted list in increasing numerical order, the rank of each element will just be its actual position in the list. Finally, the algorithm uses the rank of each element to place it in its proper sorted position.

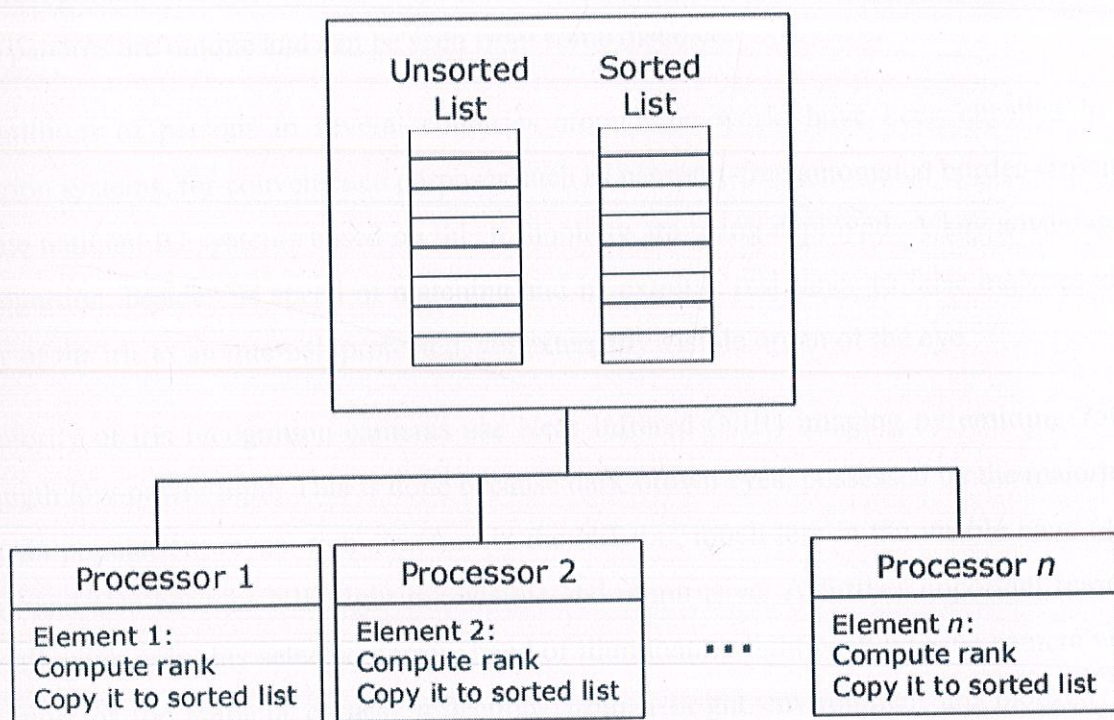


Figure 5.2: Parallel Rank Sort

Chapter 6: Iris Recognition

Image matching is a very important concept in image processing. Its applications include biometric authentication, face-recognition etc. It has a vast scope for parallelization as every pixel is manipulated separately and hence can be done parallelly. Our algorithm is based on the Euclidean Distance Classifier. Euclidean distance is simply the distance between two points. It is calculated as

$$D(p,q) = \sqrt{((q_1 - p_1)^2 + (q_2 - p_2)^2)}$$

Iris recognition is an automated method of biometric identification that uses mathematical pattern-recognition techniques on images of the iris of an individual's eyes, whose complex random patterns are unique and can be seen from some distance.

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.

The majority of iris recognition cameras use Near Infrared (NIR) imaging by emitting 750nm wavelength low-power light. This is done because dark-brown eyes, possessed by the majority of the human population, reveal rich structure in the NIR but much less in the visible band (400 - 700nm), and also because NIR light is invisible and unintrusive. A further important reason is that by allowing only this selected narrow band of illuminating light back into the camera via its filters, most of the ambient corneal reflections from a bright environment are blocked from contaminating the iris patterns

Advantages

The iris of the eye is considered to be the ideal part of the human body for biometric identification for several reasons:

- It is an internal organ & thus it is well protected against damage and wear by a highly transparent and sensitive membrane (the cornea). This distinguishes it from fingerprints, which can be difficult to recognize after years of certain types of manual labor.
- The iris is mostly flat, and its geometric configuration is only controlled by two complementary muscles (the sphincter pupillae and dilator pupillae) that control the diameter of the pupil. This makes the iris shape far more predictable than, for instance, that of the face.
- The iris has a fine texture that, like fingerprints, is determined randomly during embryonic gestation. Like the fingerprint, it is very hard (if not impossible) to prove that the iris is unique. However, there are so many factors that go into the formation of these textures (the iris and fingerprint) that the chance of false matches for either is extremely low. Even genetically identical individuals have completely independent iris textures.
- An iris scan is similar to taking a photograph and can be performed from about 10 cm to a few meters away. There is no need for the person being identified to touch any equipment that has recently been touched by a stranger, which eliminates the objection that has been raised in some cultures against fingerprint scanning, where a finger has to touch a surface, or retinal scanning, where the eye must be brought very close to an eyepiece (like looking into a microscope).
- The intricate textures are remarkably stable over many decades and are not even susceptible to the medical and surgical procedures that can affect the shape & colour of the iris. Some iris identifications have succeeded over a period of about 30 years.

Shortcomings

However, this method has the following shortcomings:

- Many commercial Iris scanners can be easily deceived by a high quality image of an iris or face in place of the real thing.

- The scanners are usually very difficult to adjust and can become bothersome for multiple people of different heights to use in succession.
- Changes in lighting can also affect the accuracy of the Scanners.
- Iris scanners are significantly expensive than some other forms of biometrics or passwords.
- The Law enforcements & immigration authorities have already made a substantial investment in finger print recognition and this technology is incompatible with the existing hardware.
- As in case of other photographic biometric technologies, iris recognition is also prone to poor image quality, with associated failure to enroll rates.

Security considerations

Methods that have been suggested to provide some defense against the use of fake eyes and irises include:

- Changing ambient lighting during the identification (switching on a bright lamp), such that the pupillary reflex can be verified and the iris image be recorded at several different pupil diameters
- Using spectral analysis instead of merely monochromatic cameras to distinguish iris tissue from other material
- Observing the characteristic natural movement of an eyeball (measuring nystagmus, tracking eye while text is read, etc.)
- Testing for retinal retroreflection (red-eye effect)
- Testing for reflections from the eye's four optical surfaces (front and back of both cornea and lens) to verify their presence, position and shape
- Using 3D imaging (e.g., stereo cameras) to verify the position and shape of the iris relative to other eye features

The Approach

The first step in the process of Iris matching is iris image acquisition. This is generally done by near infrared cameras. For the purpose of simplicity, we are using an Iris database. While opening the database we are forcing the images to grayscale because the color of the eye is insignificant in iris recognition. The patterns are more significant than the color of the eye. Next we find the mean image of the database by parallelly finding the mean of corresponding pixels in the database images. Then we center each image in the database by subtracting the corresponding pixel values of the mean image from each database image. This gives us the centered database. Similarly the input image is also centered. The next step is comparing the centered input image and the centered database using the Euclidean distance classifier method. The image with the least Euclidean distance is checked against a threshold value. If the value is less than the threshold the matched image is displayed.

Pseudocode

```
for each thread t, in block b
    take corresponding pixels from each image, find mean.
for each thread t, in block b
    subtract corresponding mean values from pixels of database images.
for each thread t, in block b
    subtract corresponding mean values from pixel values of input image.
for each thread t, in block b
    subtract each centered database image from the centered input image.
calculate total difference between input image and each database image.
find matched image according to the threshold value
```

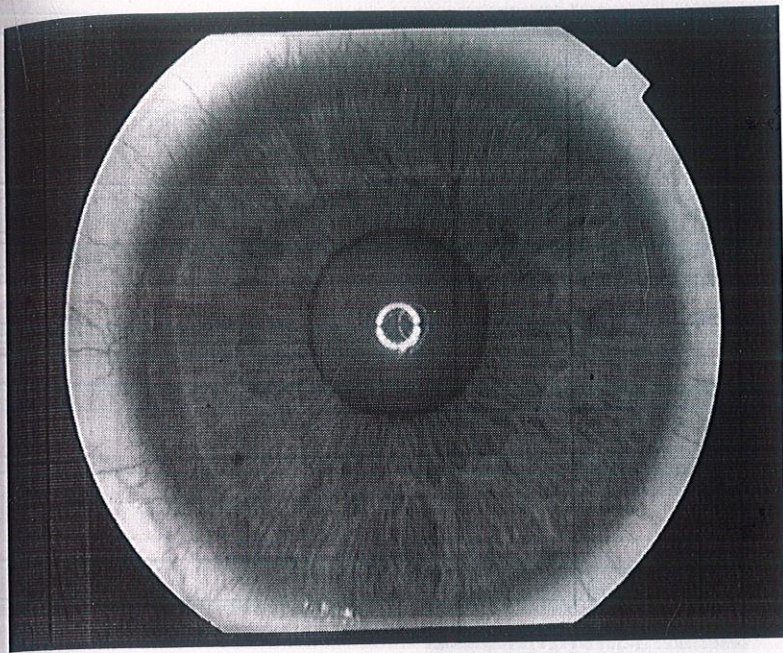



Figure 6.1: Test Image

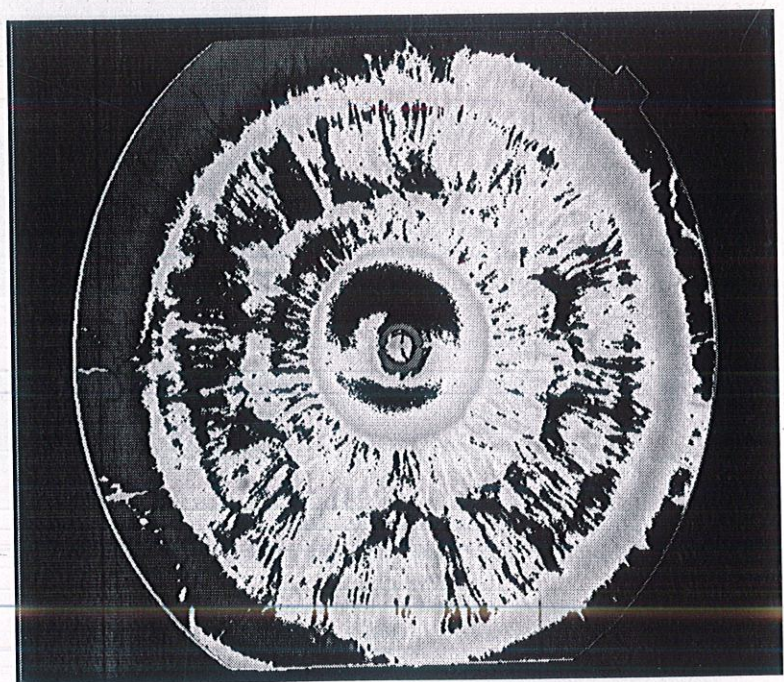


Figure 6.2: Image Centered about the mean

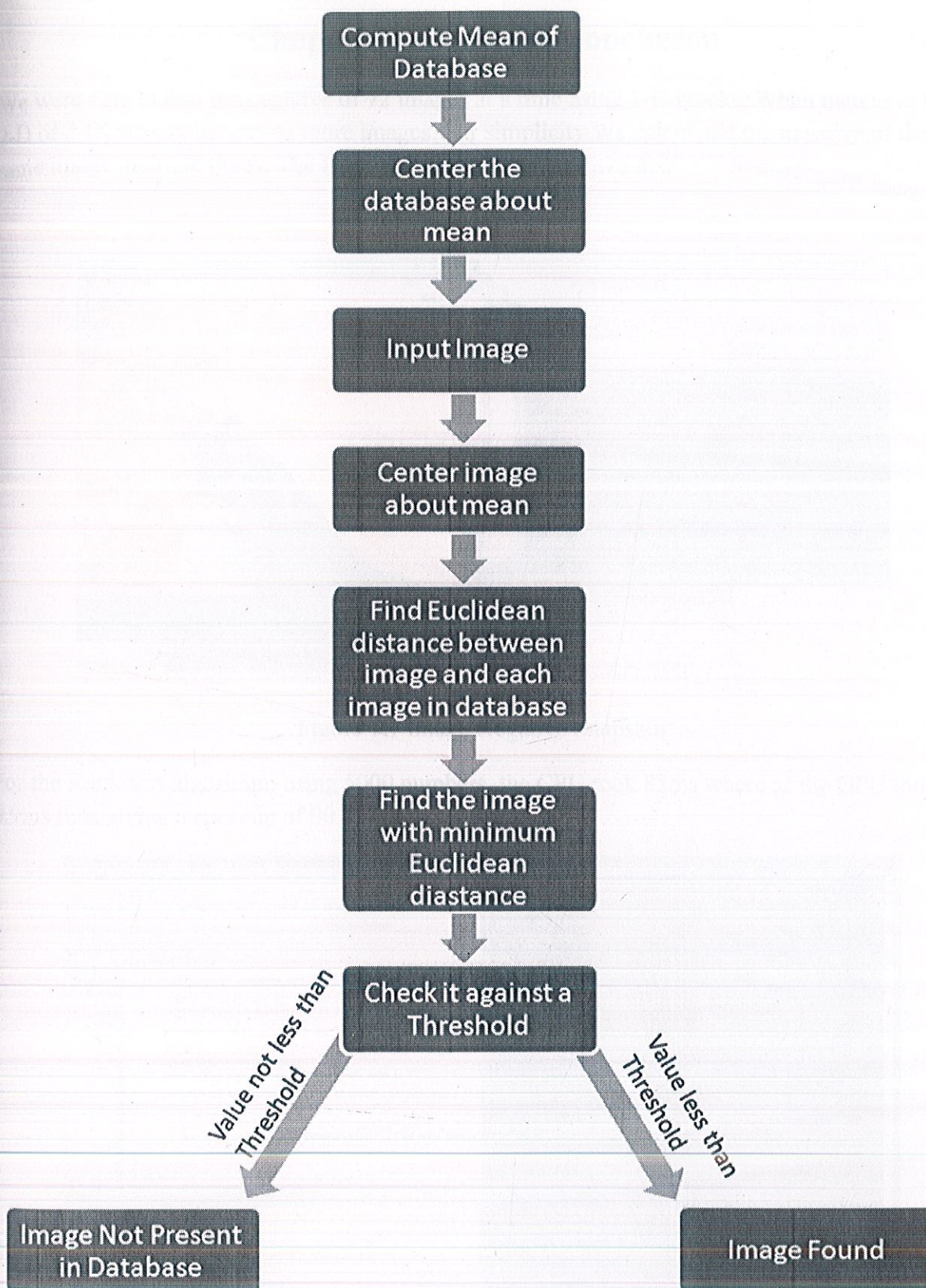


Figure 6.3: Iris Recognition Design Flow

Chapter 7: Results & Conclusion

We were able to find the negative of 72 images at a time using 1-D blocks. When increased to 2-D or 3-D, we can have even more images. For simplicity We calculated the negative of the same image multiple times. The Parallel algorithm took 435.75 ms.

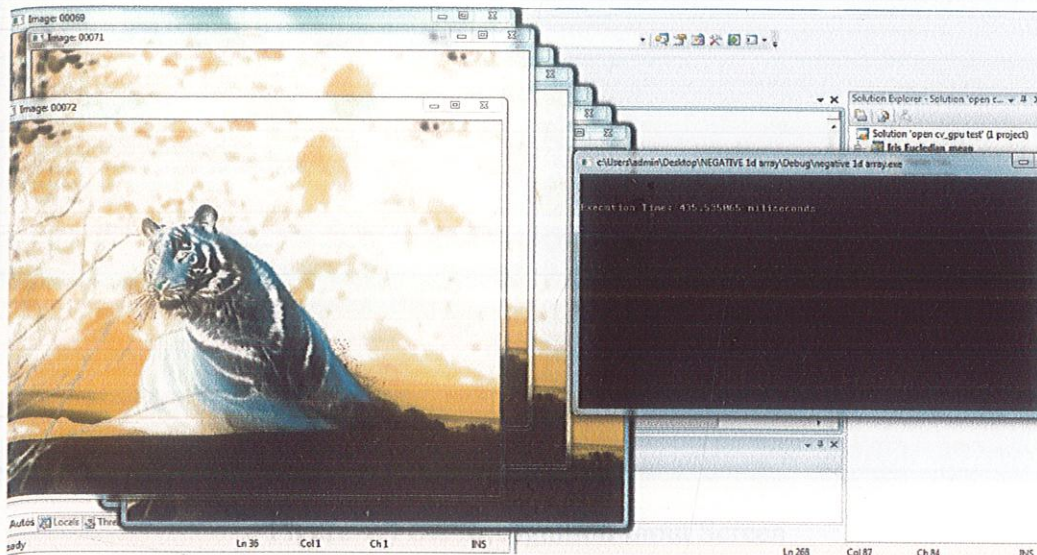


Figure 7.1 Image Negative Snapshot

For the Rank sort algorithm, using 5000 numbers, the CPU took 82ms where as the GPU took 0.9ms thus giving a speedup of 90.7x.

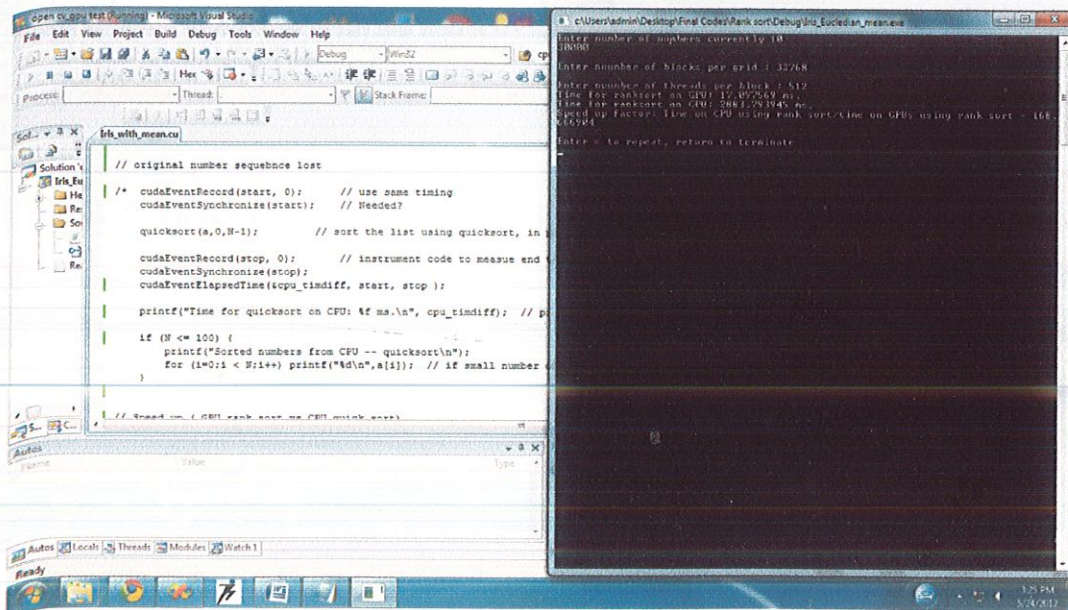


Figure 7.2 Rank Sort Snapshot

Considerable speed up was observed for iris recognition also.

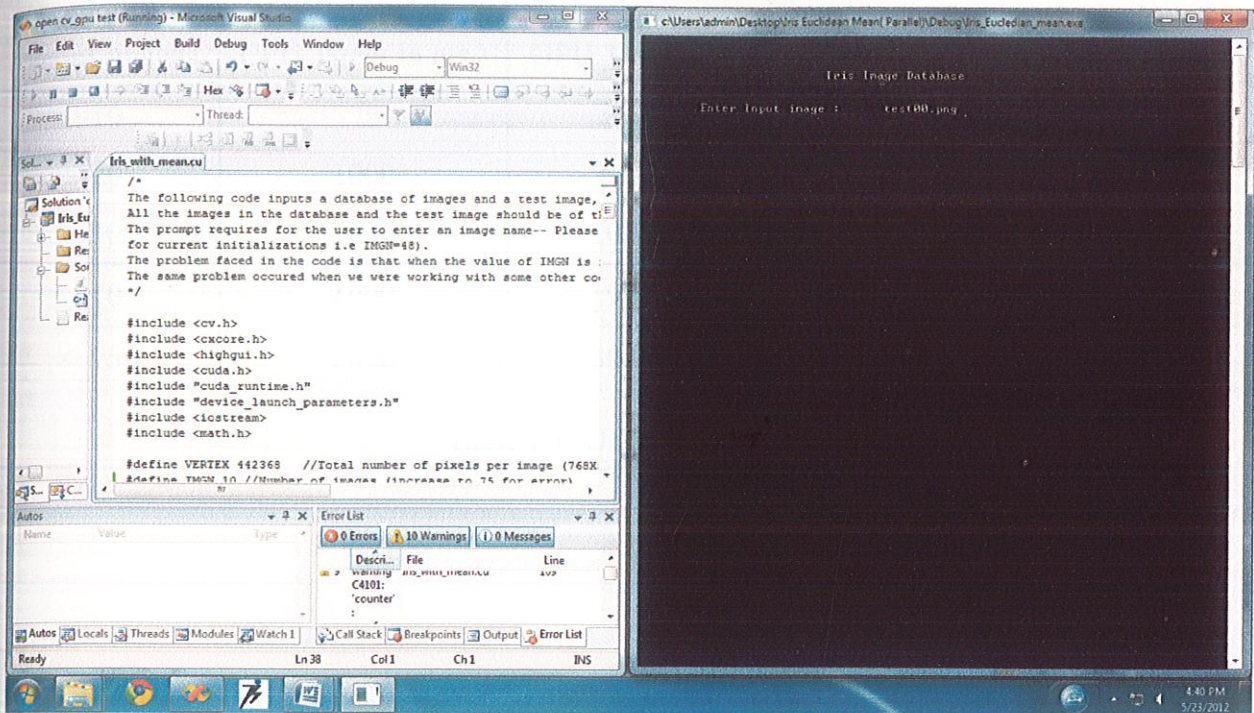


Figure 7.3 Iris Recognition Input Screen

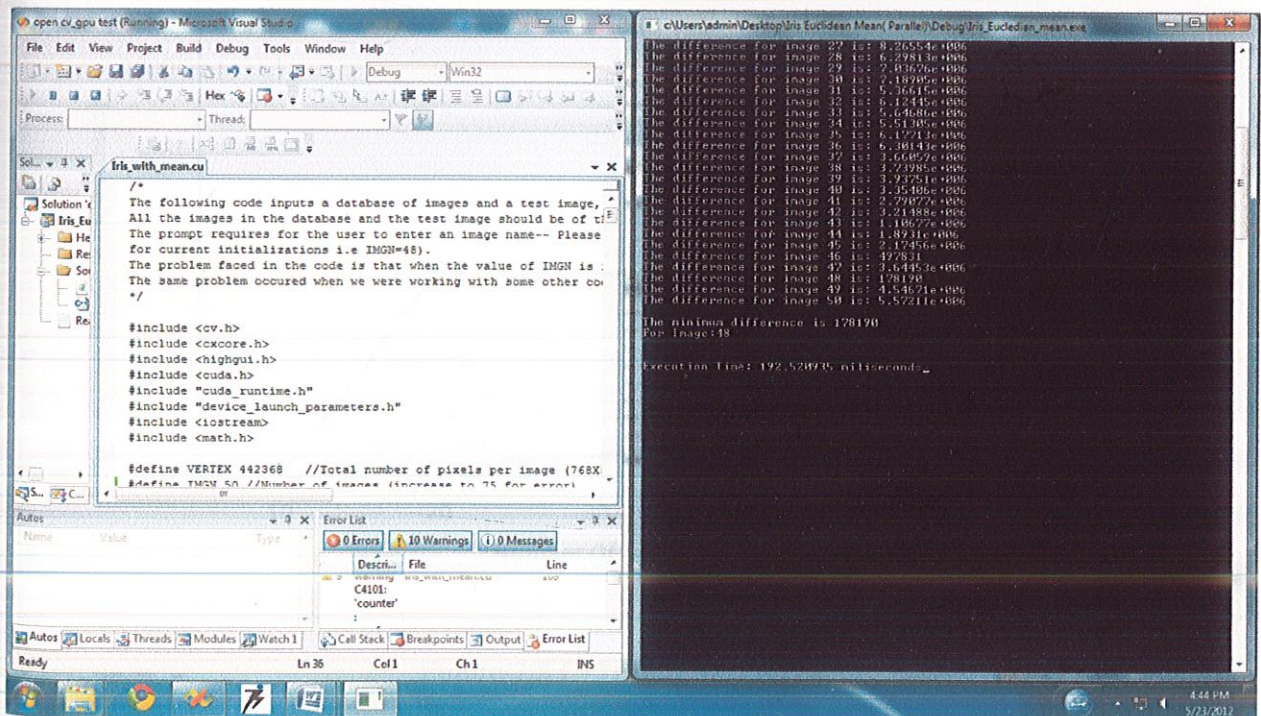


Figure 7.4 Iris Recognition Output Screen

The table below shows the speedup observed for iris recognition by using parallel processing when compared with a similar serial algorithm.

No. of Images	CPU Time(in ms.)	GPU Time(in ms.)	Speedup (CPU Time/ GPU time)
10	0	31.3702	0
20	1000	61.5245	16.25368756
30	1000	107.3763	9.313042077
50	1000	192.5209	5.194241249
75	2000	302.0339	6.621773251

Table 7.1: Observed Speedup

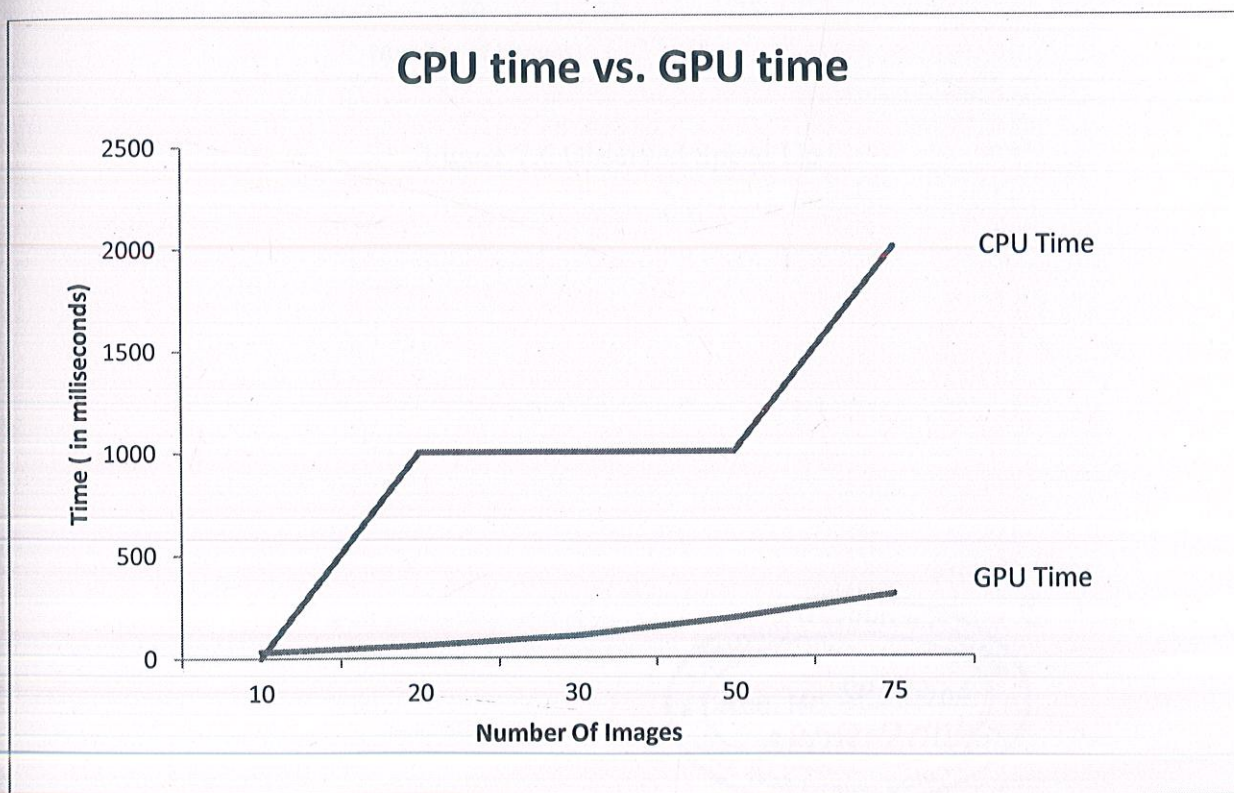


Figure 7.5 CPU time vs. GPU time

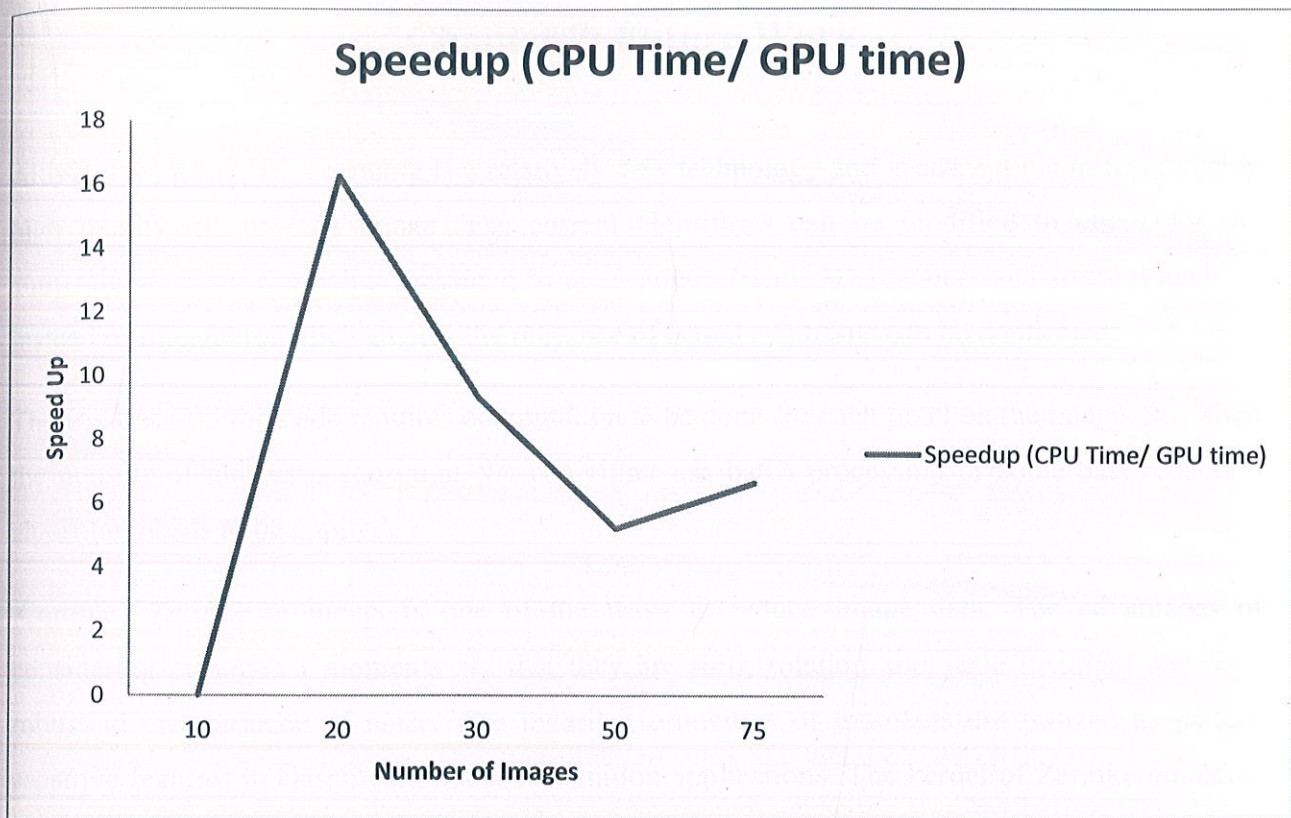
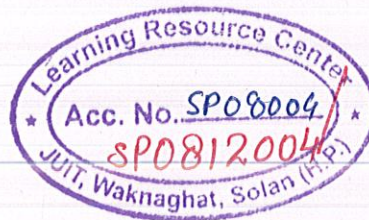


Figure 7.6 Observed Speed Up



Chapter 7: Future Work

Although efficient, iris scanning is a relatively new technology and is susceptible to frauds using high quality iris or face image. The current algorithms can be modified to check for the authenticity of the eye being presented to the scanner. Using 3D scanners and spectral analysis instead of monochromatic cameras the presence of actual eye tissue can be confirmed.

The Iris Recognition code requires computation to be done for each pixel on the image. So, when the number of images is too large, we can either use batch processing or some data reduction algorithm needs to be applied.

Complex Zernike moments is one of the ways to reduce image data. The advantages of considering orthogonal moments are that they are shift, rotation and scale invariant and very robust in the presence of noise. The invariant properties of moments are utilized as pattern sensitive features in classification and recognition applications. The kernel of Zernike moments is a set of orthogonal Zernike polynomials defined over the polar coordinate space inside a unit circle. Applying this algorithm can considerably increase the efficiency of the program.

References

- M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, CUDASA: Compute Unified Device and Systems Architecture
- VAN DEN ELSEN P. A., POL E.-J. J., VIERGEVER M. A.: Medical image matching: A review with classification. IEEE Engineering in Medicine and Biology 12 (1993),
- Accurate and Efficient Computation of High Order Zernike Moments, Gholam Reza Amayeh, Ali Erol, George Bebis, and Mircea Nicolescu Computer Vision Laboratory, University of Nevada, Reno, NV 89557
- Zernike Moments for Facial Expression Recognition Seyed Mehdi Lajevardi, Zahir M. Hussain School of Electrical & Computer Engineering, RMIT University, Melbourne, Australia
- Sketched Symbol Recognition using Zernike Moments Heloise Hse and A. Richard Newton Department of Electrical Engineering and Computer Sciences University of California at Berkeley
- Programming massively parallel processors, by David B. Kirk & Wen-Mei W. HWU
- Cuda at Parallel Paronama
- Whitepaper: Fermi Architecture
- <http://developer.nvidia.com/category/zone/cuda-zone>
- <http://en.wikipedia.org/wiki/CUDA>
- http://en.wikipedia.org/wiki/Iris_recognition
- en.wikipedia.org/wiki/Iris_recognition
- www.cl.cam.ac.uk/~jgd1000/ Iris recognition – University of Cambridge
- http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/SHUTLER3/node11.html
- <http://www.cse.iitk.ac.in/users/biometrics/pages/iris.htm>

CODE

A. Serial Iris Recognition

/* The following code inputs a database of images and a test image, and matches them using the euclidian classifier.

All the images in the database and the test image should be of the same dimensionality.

The prompt requires for the user to enter an image name-- Please enter the name as "img0XX.png".

*/

// Header files

#include "stdafx.h"

#include <cv.h>

#include <cxcore.h>

#include <highgui.h>

#include <iostream>

#include<ctime>

#define VERTEX 442368

#define IMGN 790

#define SIZE1 768

#define SIZE2 576

using namespace std;

// Main Function


```

int _tmain(int argc, _TCHAR* argv[])
{
    IplImage *imgp, *imgdbp[IMGN];

    float *img, *dbimg, *mean, sum[IMGN], mini;

    img=(float*)malloc(VERTEX*sizeof(float));

    dbimg=(float*)malloc(IMGN*VERTEX*sizeof(float));

    mean=(float*)malloc(VERTEX*sizeof(float));

    char junk, ipimg[20];

    int i, j;

    int n=2, m=0;

    CvMat *mat1[IMGN], *mat;

    char name[] = "img000.png";

    name[10]='\0';

    cout<<"\n\n\n \t\t\t Iris Image Database";

    cout<<"\n\n\n \tEnter Input image :\t ";

    cin>>ipimg;

    imgp = cvLoadImage(ipimg, 0);

    mat = cvCreateMat(imgp->height, imgp->width, CV_8UC1 );

    cvConvert( imgp, mat );

    int c=0;

    for(long i=0; i<SIZE2; i++)

    {

        for(int j=0; j<SIZE1; j++)

```



```

    {

        CvScalar scal = cvGet2D( mat,i,j);

        img[c] = scal.val[0];

        c=c+1;

    }

}

c=0;

j=0;

// Inputing images from database

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

{

    j++;

    switch(name[5])

    {

        case '9': switch(name[4])

        {

            case '9':

                switch(name[9])

                {

                    case '9':

                        name[5]='0';name[4]='0';name[3]='0';

                        break;

```



```

        default : name[5]='0';

        name[4]='0';

        name[3]++;

    }break;

    default : name[5]='0'; name[4]++;break;

    }

    break;

    default : name[5]++;break;

}

if(j>48)

{

    name[3]='0';

    name[4]='0';

    name[5]='1';

    j=0;

}

imgdbp[i] = cvLoadImage(name,0);

mat1[i] = cvCreateMat(imgdbp[i]->height,imgdbp[i]-
->width,CV_8UC1 );

cvConvert( imgdbp[i], mat1[i] );

}

// Converting images to 1-D Arrays

```



```

for(int m=0;m<IMGN;m++)

{

    for(long i=0;i<SIZE2;i++)

    {

        for(int j=0;j<SIZE1;j++)

        {

            CvScalar scal = cvGet2D( mat1[m],i,j);

            dbimg[c] = scal.val[0];

            c=c+1;

        }

    }

}

time_t t,t2;

t=time(&t);

// Calculating the Mean

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

{

    mean[i]=dbimg[i];

    for(j=1;j<IMGN;j++)

    {

        mean[i]+=dbimg[i+(VERTEX*j)];

```



```

    }

    mean[i]=mean[i]/IMGN;

}

// Subtracting Mean from Database
for(i=0;i<VERTEX*IMGN;i++)
{
    dbimg[i]=dbimg[i]-mean[i%VERTEX];
}

// Subtracting Mean from Input Image
for(i=0;i<VERTEX;i++)
{
    img[i]=img[i]-mean[i];
}

// Subtracting Input image from each database image
for(i=0;i<VERTEX*IMGN;i++)
{
    dbimg[i]=dbimg[i]-img[i%VERTEX];
}

// Calculating the total difference between
// Input & Database images.
for(i=0;i<IMGN;i++)

```



```

{

    sum[i]=0;

    for(j=i*VERTEX;j<(i*VERTEX)+VERTEX;j++)

    {

        sum[i]+=dbimg[j];

    }

    if(sum[i]<0)

        sum[i]*=-1;

    cout<<endl<<"The difference for image "<<i+1<<" is:
"<<sum[i];

}

// Calculating the minimum difference

mini=sum[0];

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

{

    if(mini>sum[i])

    {

        mini=sum[i];

    }

}

cout<<endl<<"\nThe minimum difference is "<<mini;

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

```



```

{

    if(sum[i]==mini)

        cout<<endl<<"For Image:"<<i+1;

}

t2=time(&t2);

float rtim=difftime(t2,t);

printf("\n\nExecution Time: %f second(s)",rtim); // Print
Elapsed time

cin>>i;

return 0;

}

```

B. Parallel Iris Recognition

/*

The following code inputs a database of images and a test image, and matches them using the euclidian classifier.

All the images in the database and the test image should be of the same dimensionality.

The prompt requires for the user to enter an image name-- Please enter the name as "img0XX.png".

*/


```

#include <cv.h>

#include <cxcore.h>

#include <highgui.h>

#include <cuda.h>

#include "cuda_runtime.h"

#include "device_launch_parameters.h"

#include <iostream>

#include <math.h>


#define VERTEX 442368 //Total number of pixels per image (768X576)

#define IMGN 64

#define SIZE1 768 //Dimensionality of
#define SIZE2 576 //the images
#define NUMBER 864 // Vertex/512 (Used for the kernel
doing the addition)

using namespace std;


// This function finds the mean of the database.

// mean is calculated as the mean value of each pixel.

// thereby generating a mean image of same dimentionalitiy

```



```
_global__ void devmean(float *devdb, float *mn)
```

```
{  
  
    int tx=threadIdx.x;  
  
    int bx=blockIdx.x;  
  
    long id=(bx*blockDim.x)+tx;  
  
    for(int i=0;i<IMGN;i++)  
  
        mn[id]+=devdb[id+VERTEX*i];  
  
  
    mn[id]=mn[id]/IMGN;  
  
}
```

```
// Input image is centered by subtracting it from mean
```

```
_global__ void devcenterimg(float *devimg, float *mn, long  
threads)
```

```
{  
  
    int tx=threadIdx.x;  
  
    int bx=blockIdx.x;  
  
    long id=bx*blockDim.x+tx;  
  
    if(id<threads)  
  
    {  
  
        devimg[id]=devimg[id]-mn[id];  
  
    }  
  
}
```



```
}  
  
// centering of database with the mean
```

```
__global__ void devcenter(float *devdb, float *mn, long threads)
```

```
{
```

```
int tx=threadIdx.x;
```

```
int bx=blockIdx.x;
```

```
long id=bx*blockDim.x+tx;
```

```
__shared__ float smn[1024/IMG_N];
```

```
if(id%IMG_N==0)
```

```
{
```

```
    smn[tx/IMG_N]=mn[id/IMG_N];
```

```
}
```

```
__syncthreads();
```

```
if(id<threads)
```

```
devdb[((id%IMG_N)*(threads/IMG_N)+(id/IMG_N)]=devdb[((id%IMG_N)*(th  
reads/IMG_N)+(id/IMG_N)]-smn[tx/IMG_N];
```



```
// Parallel addition program to calculate the sum of differences
```

```
__global__ void add(float *devdb, float *devsum)
```

```
{
```

```
    int tx=threadIdx.x;
```

```
    int bx=blockIdx.x;
```

```
    long id=bx*blockDim.x+tx;
```

```
    __shared__ float tempsum[512];
```

```
    tempsum[tx]=devdb[id];
```

```
    __syncthreads();
```

```
    for(int i=512/2;i>0;i=i/2)
```

```
    {
```

```
        if(tx<i)
```

```
            tempsum[tx]+=tempsum[tx+i];
```

```
            __syncthreads();
```

```
    }
```

```
    if(tx==0)
```

```
    {
```

```
        devsum[bx]=tempsum[tx];
```

```
    }
```



```

// main
int main()
{
    // Variable declarations
    IplImage *imgp,*imgdbp[IMG_N];

    int i,j;

    float *img, *dbimg;

    img=(float*)malloc(VERTEX*sizeof(float));

    dbimg=(float*)malloc(IMG_N*VERTEX*sizeof(float));

    char junk,ipimg[20];

    CvMat *mat1[IMG_N],*mat;

    char name[]="img000.png";

    name[10]='\0';

    float *sum;

    int imageno=0;

    // Entering The test image

    cout<<"\n\n\n \t\t\t Iris Image Database";

    cout<<"\n\n\n \tEnter Input image :\t ";

    //Loading the test image into float array, to be matched

```



```

cin>>ipimg;

imgp = cvLoadImage(ipimg,0);

int k,counter;

mat = cvCreateMat(imgp->height,imgp->width,CV_8UC1 );

cvConvert( imgp, mat );

int c=0;

for(long i=0;i<SIZE2;i++)
{
    for(int j=0;j<SIZE1;j++)
    {
        CvScalar scal = cvGet2D( mat,i,j);

        img[c] = scal.val[0];

        c=c+1;
    }
}

// Load The Database

c=0;

j=0;

for(i=0;i<IMGN ;i++)
{

```



```

j++;

switch(name[5])

{

    case '9': switch(name[4])

    {

        case '9':

            switch(name[3])

            {

                case '9': name[5]='0';name[4]='0';name[3]='0';

                break;

                default : name[5]='0';

                name[4]='0';

                name[3]++;

            }break;

            default : name[5]='0'; name[4]++;break;

        }

        break;

        default : name[5]++;break;

    }

    if(j>48)

```



```

{
    name[4]='0';
    name[5]='1';
    name[10]='\0';
    j=1;
}

imgdbp[i] = cvLoadImage(name,0);
mat1[i] = cvCreateMat(imgdbp[i]->height,imgdbp[i]->width,CV_8UC1
);
cvConvert( imgdbp[i], mat1[i] );
}

for(int m=0;m<IMGN;m++)
{
    for(long i=0;i<SIZE2;i++)
    {
        for(int j=0;j<SIZE1;j++)
        {
            CvScalar scal = cvGet2D( mat1[m],i,j);
            dbimg[c] = scal.val[0];
            c=c+1;
        }
    }
}

```



```

    }

}

//CUDA kernel calls

float *devimg;

float *devdb;

float *devsum;

float *mn;

long threads, blocks;


cudaMalloc((void**)&devimg, VERTEX*sizeof(float));

cudaMalloc((void**)&devdb, IMGN*VERTEX*sizeof(float));

cudaMalloc((void**)&mn, VERTEX*sizeof(float));

cudaMemcpy(devdb, dbimg, IMGN*VERTEX*sizeof(float),
cudaMemcpyHostToDevice);

cudaMemcpy(devimg, img, VERTEX*sizeof(float),
cudaMemcpyHostToDevice);

cudaEvent_t start, stop;

// Generate events

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

threads=VERTEX;

```



```

blocks=threads%1024==0?threads/1024:threads/1024+1;

// Calculating the mean of the images
devmean<<<blocks,1024>>>(devdb,mn);

threads=VERTEX;

blocks=threads%1024==0?threads/1024:threads/1024+1;

// Centering the test image
devcenterimg<<<blocks,1024>>>(devimg,mn,threads);

threads=VERTEX*IMGN;

blocks=threads%1024==0?threads/1024:threads/1024+1;

// Centering the database images
devcenter<<<blocks,1024>>>(devdb,mn,threads);

threads=VERTEX*IMGN;

blocks=threads%1024==0?threads/1024:threads/1024+1;

// Calculating Euclidian distance
devcenter<<<blocks,1024>>>(devdb,devimg,threads);

blocks=VERTEX%512==0?VERTEX/512:(VERTEX/512)+1;

blocks=blocks*IMGN;

```



```

cudaMalloc((void**)&devsum, blocks*sizeof(float));

//Calculating the sum of elements present in each individual
block

add<<<blocks,512>>>(devdb, devsum);

sum=(float*) malloc(blocks*sizeof(float));

cudaMemcpy(sum, devsum, blocks*sizeof(float),
cudaMemcpyDeviceToHost);

cudaMemcpy(img, devimg, VERTEX*sizeof(float),
cudaMemcpyDeviceToHost);

cudaFree(devimg);

cudaFree(devdb);

cudaFree(devsum);

cudaFree(mn);

//CUDA part end

// Adding the final sums of the blocks that represent the image.
for (i=0;i<IMGN;i++)
{
    for(j=(i*NUMBER)+1;j<(i*NUMBER)+NUMBER;j++)

        sum[i*NUMBER]+=sum[j];

    if(sum[i*NUMBER]<0)

        sum[i*NUMBER]*=-1;
}

```



```

        cout<<endl<<"The difference for image "<<i+1<<" is:
"<<sum[i*NUMBER];

    }

    //Calculating the minimum of all the distances present.
    float minimumval=sum[0*NUMBER];

    for(int q=0;q<IMGN;q++)
    {

        if(sum[q*NUMBER]<minimumval)

        {

            minimumval=sum[q*NUMBER];

            imageno=q;

        }

    }

    cout<<endl<<"\nThe minimum difference is "<<minimumval;

    for(i=0;i<IMGN;i++)
    {

        if(sum[i*NUMBER]==minimumval)

            cout<<endl<<"For Image:"<<i+1;

    }

    cudaEventRecord(stop, 0); // Trigger Stop event

    cudaEventSynchronize(stop); // Sync events (BLOCKS till last
(stop in this case) has been recorded!)

```



```
float elapsedTime; // Initialize elapsedTime;

cudaEventElapsedTime(&elapsedTime, start, stop); // Calculate
runtime, write to elapsedTime -- cudaEventElapsedTime returns

printf("\n\n\nExecution Time: %f milliseconds", elapsedTime); //
Print Elapsed time

cin>>junk;

return (0);
```