Parallel Implementation of Similarity Measures on GPU Architecture using CUDA

Kuldeep Yadav

Department of Computer Science and Engineering
College of Engineering Roorkee, Roorkee-247667, INDIA

Kul82_deep@rediffmail.com

Ankush Mittal

Department of Computer Science and *Engineering*

College of Engineering Roorkee, Roorkee-247667, INDIA

dr.ankush.mittal@gmail.com[2]

M.A Ansari

Department of Electrical and Engineering Gautam Budha University, Greater Noida, INDIA

ma.ansari@.ieee.org

VennkteshVishwarup

Department of Computer Science and Engineering

College of Engineering Roorkee, Roorkee-247667, INDIA

vnktshv@rediffmail.com

Abstract- Image processing and pattern recognition algorithms take more time for execution on a single core processor. Graphics Processing Unit (GPU) is more popular now-a-days due to their speed, programmability, low cost and more inbuilt execution cores in it. Most of the researchers started work to use GPUs as a processing unit with a single core computer system to speedup execution of algorithms and in the field of Content based medical image retrieval (CBMIR), Euclidean distance and Mahalanobis plays an important role in retrieval of images. Distance formula is important because it plays an important role in matching the images. In this research work, we parallelized Euclidean distance algorithm on CUDA. CPU with Intel® Dual-Core E5500 @ 2.80GHz and 2.0 GB of main memory which run on Windows XP (SP2). The next step was to convert this code in GPU format i.e. to run this program on GPU NVIDIA GeForce series 9500GT model having 1023 MB of video memory of DDR2 type and bus width of 64bit. The graphic driver we used is of 270.81 series of NVIDIA. In this paper both the CPU and GPU version of algorithm is being implemented on the MATLAB R2010. The CPU version of the algorithm is being analyzed in simple MATLAB but the GPU version is being implemented with the help of intermediate software Jacket-win-1.3.0. For using Jacket, we have to make some changes in our source code so to make the CPU and GPU to work simultaneously and thus reducing the overall computational acceleration . Our work employs extensive usage of highly multithreaded architecture of multi-cored GPU. An efficient use of shared memory is required to optimize parallel reduction in Compute Unified Device Architecture (CUDA), Graphic Processing Units (GPUs) are emerging as powerful parallel systems at a cheap cost of a few thousand rupees.

Keywords: Euclidean distance, Mahalanobis Distance, Content Based Medical Image Retrieval (CBMIR), CUDA, GPU, Parallelization

1. INTRODUCTION

Content-based image retrieval (CBIR) has gained considerable attention especially in the last decade. Image retrieval based on content is extremely useful in several applications such as medicine, publishing and advertising, historical research, fashion and graphic design, architectural and engineering design, crime prevention etc [1-4]. In this paper we are focusing on medical Image retrieval and similarity measures. Numerous commercial and experimental CBIR systems are now available e.g. IBM's QBIC (Query by Image Content), Virage's VIR Image Engine, Excalibur's Image Retrieval Ware, or Columbia University's Web SEEK. Also, many web search engines are now equipped with CBIR facilities, for example Alta Vista.

Due to advances in data storage and rapid growth of Medical imaging along with image acquisition technologies, there is creation of huge image data sets and digital archives. Medical Images as a norm are now being stored in compressed domain so as to collect the constraints imposed by storage and transmission costs of managing large amount of image data. Effective and efficient image indexing and accessing tools are much importance in order to fully utilize the huge digital data available. Even though so much effort has been spent on image retrieval techniques, one important aspect of image compression has been neglected almost completely. As the amount of image data is ever increasing, where as hardware and network resources are still Content based image retrieval (CBIR) refers to a process of retrieving expected images from databases according to image queries or a number of features (e.g. colour, texture and shape), which can be automatically extracted from the images using reasoning techniques.

2. EUCLIDEAN DISTANCE FORMULA

[5] A central problem in image recognition and computer vision is determining the distance between images. Considerable efforts have been made to define image distances that provide intuitively reasonable results. Among all the image metrics, Euclidean distance is the most commonly used due to its simplicity.[6] Let x, y be two M by N images, $x = (x^1, x^2 ,---, x^{MN})$, $y = (y^1, y^2 ,----, y^{MN})$, where x^{kN+l} , y^{kN+l} are the gray levels at location (k,l) . The Euclidean distance (x, y) is given by:

$$d^2_E (x,y)= \Sigma^{MN}_{k=1} (x^k - y^k)^2 \qquad (1)$$

The Euclidean distance gives counter intuitive result. The pair with more similarity has a larger Euclidean distance! This phenomenon is caused by the fact that the Euclidean distance defined in (1) does not take into account that x, y are images, xk, yk are gray levels on pixels. For images, there are spatial relationships between pixels.

While the Euclidean distance metric is popular with in the multimedia indexing community, it is by no means the perceptually "correct" distance measure. Hence, significant research activity (in content-based image retrieval) has been directed toward Mahalanobis (or weighted Euclidean) distances (see [8]). The Mahalanobis distance measure has more degrees of freedom than the Euclidean distance and by proper updating (or relevance feedback), has been found to be a much better estimator of user perceptions [7]. Let $d_W(x, y) = (x - y)^{TW}(x - y)$ be the distance between any two feature vectors x and y. Without loss of generality, we assume W is symmetric and positive definite i.e. $dW(\cdot, \cdot)$ is a metric. Let $H(a, b) = \{x: aT x + b = 0\}$ be a hyper plane and y a point in the space outside of it. Then, $d_W(y,H) = \min_{x\in H} d_W(x, y) =(\min_{x\in H} d_W(x, y)^2)^{1/2}$

As we all know the Euclidean distance is:

$$d^2 (p_1, p_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2. \qquad (2)$$

We can reformulate this distance function as an inner-product in vector space:

$$d^2 (p_1, p_2) = (p_1 - p_2)^T (p_1 - p_2), \qquad (3)$$

Where T represents matrix transpose. ok , so far no problems. What if we can't actually directly observe X and Y,But can only observe scaled versions, s1X and s2Y, where s1, s2 6= 0, but are otherwise arbitrary. Let's indicate a scaled observation point as p'_i. Can we still calculate $d^2 (p_1, p_2)$?

$$\begin{aligned} d^2 (p_1, p_2) &= (x_1 - x_2)^2 + (y_1 - y_2)^2 \\ &= (x'_1/s_1 - x'_2/s_1)^2 + (y'_1/s_2 - y'_2/s_2)^2 \\ &= (x'_1 - x'_2)^2/s^2_1 + (y'_1 - y'_2)^2/s^2_2. \end{aligned}$$

So as long as we know the scale factor, we can calculate d2. Well, it's pretty easy to calculate the scale factor and there are many ways to do it. For reasons that should become clear below, let's use the variance of each of the variables as a measure of spread. It is a basic fact from statistics that Var (sX) = sVar (X), and since, by definition, Var (X) = 1, we have Var (X′) = s and we can calculate the scale factor for each variable and subsequently the distance in the undistorted space. Now, let's try to be a bit trickier. What if we can't directly observe X and Y, but can only observe linear combinations:

$$X' = {}_{a11}X + {}_{a12}Y$$
$$Y' = {}_{a21}X + {}_{a22}Y. \quad (4)$$

Can we still calculate d^2 (p_1, p_2)? First, let's write equation 3 in a more convenient form:

$$p' = Ap, \quad (5)$$

Where A = [a11 a12; a21 a22]. We could have done a similar thing in the previous paragraph, but I think it's more intuitive the way it is. So, our Euclidean distance formula becomes:

$$d^2 (p_1, p_2) = (A^{-1} (p'_1 - p'_2))^T (A^{-1} (p'_1 - p'_2))$$
$$= (p'_1 - p'_2)^T A^{-1\,T} A^{-1} (p'_1 - p'_2)$$

If we do a bit of manipulation and actually calculate the matrix $A^{-1T} A^{-1}$, then we get:

$$A^{-1T} A^{-1} = 1/ (a_{12}a_{21} - a_{11}a_{22})^2 \, [a^2_{21} + a^2_{22} -a_{11}a_{21} - a_{12}a_{22}; -a_{11}a_{21} - a_{12}a_{22} \; a^2_{11} + a^2_{12}] \quad (6)$$

Now let's take a bit of a side-track and calculate the covariance matrix for X′ andY ′. First the variance of X′ and Y ′:

$$\mathrm{Var}(X') = a^2_{11} + a^2_{12}$$

Where E represents expectation. We get similar results for the variance of Y ′, and can follow a similar treatment to get the covariance of X′and Y ′:

$$\mathrm{Cov}(X', Y') = a_{11}a_{21} + a_{12}a_{22}. \quad (7)$$

This leaves us with the covariance matrix:

$$E (X', Y') = [a^2_{11} + a^2_{12} \quad a_{11}a_{21} + a_{12}a_{22};$$
$$a_{11}a_{21} + a_{12}\, a_{22} \quad a^2_{21} + a^2_{22}.] \quad (8)$$

Now if we take the inverse of E, then we get:

$$E(X', Y')^{-1} = 1/(a_{12}a_{21} - a_{11}a_{22})^2 \; [a^2_{21} + a^2_{22} -a_{11}a_{21} - a_{12}a_{22} \,; -a_{11}a_{21} - a_{12}a_{22} \;\; a^2_{11} + a^2_{12}] \quad (9)$$

Which looks surprisingly like equation 6.Thus we are led to the remarkable result that to measure the Euclidean distance in an underlying uncorrelated space, we need to use the Mahalanobis distance:

$$d^2 (p_1, p_2) = (p'_1 - p'_2)^T E^{-1}(p'_1 - p'_2) \quad (10)$$

2.1 USE OF DISTANCE FORMULA IN CBMIR

Distance formula is important for CBMIR. As because it helps in retrieval of the image equivalent to query image from the images available in Database. As a query Image is entered the distance from its points are used to match with the points on other Image to get the results in case of Euclidean distance. This process is changed a bit and repeated in the same way to match the query Image from the Database in case of the Mahalanobis Distance. So, we can get that both the distance formula plays an important role In CBMIR.

3. BACKGROUND

In this section, we have discussed about the Jacket v 1.3.0 which is a Graphics Processors for general purpose computing. Over the past few years, specialized coprocessors from floating point hardware to field programmable gate arrays have enjoyed a widening performance gap with traditional x86 based processors. Of these, graphics processing units (GPUs) have advanced at an astonishing rate, currently capable of delivering over 1 TFOPS of single precision performance and over 300 GFLOPS of double precision while executing up to 240 simultaneous threads in one low cost package. As such, GPUs have gained significant popularity as powerful tools for high performance computing (HPC) achieving 20100 times the speed of their x86 counterparts in applications such as physics simulation, computer vision, options pricing, sorting, and search. As with previous research compressed sensing studies based on Graphics Processing Units (GPUs) provide fast implementations. However, only a small number of these GPU-based studies concentrate on compressed sensing Since the GPU which we have taken (NVIDIA 9500 GT) is the most basis model has high portability and is easily available in present day laptop and desktops so can be implemented directly. However synchronizing of host and device with suitable parallel implementation is the most challenging part. Which has been parallelized by us? We have broken the process in threads of blocks and managed those threads inside a special thread managing hardware called as GPU with the help of the environment and set of libraries provided by CUDA. [9]

3.1. JACKET OVERVIEW

Jacket connects Matlab to the GPU. Matlab is a technical computing language that integrates computation, visualization and programming in an easy to use environment that has found wide popularity both in the industry and academia. It is used across the breath of technical computing applications including mathematical computations, algorithm development, data analysis, data visualization and application development. With the GPU as a backend computation engine, Jacket brings together the best of three important computational worlds: computational speed, visualization, and the user friendliness of M programming. Jacket enables developers to write and run code on the GPU in the native M language used in Matlab. Jacket accomplishes this by automatically wrapping the M language into a GPU compatible form. By simply casting input data to Jacket's GPU data structure, Matlab functions are transformed into GPU functions. Jacket also preserves the interpretive nature of the M language by providing real time, transparent access to the GPU compiler.

3.2. INTEGRATION WITH MATLAB

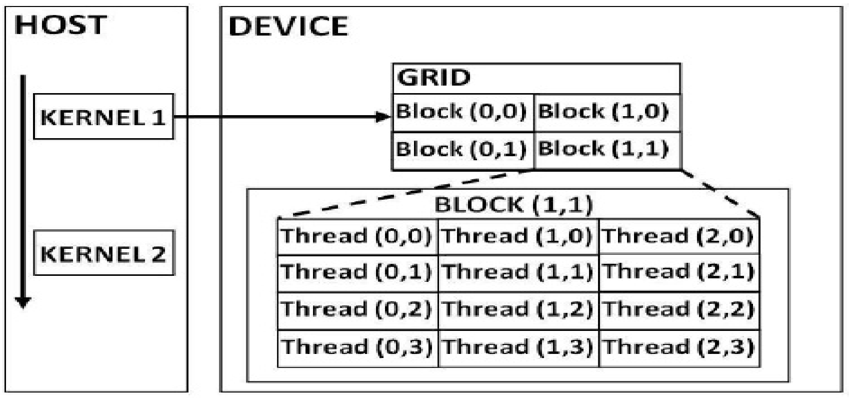
Once Jacket is installed, it is transparently integrated with the Matlab's user interface and the user can start working interactively through the Matlab desktop and command window as well as write M-functions using the Matlab editor and debugger. All Jacket data is visible in the Matlab workspace, along with any other Matlab matrices.

3.3 INTRODCTION TO NVIDIA CUDA ARCHITECTURE

CUDA™ is a general purpose parallel computing architecture introduced by NVIDIA. It contains the CUDA Instruction Set Architecture (ISA) and parallel compute engine in the GPU. The CUDA architecture is programmed using C language, which can then be run with great performance on a CUDA enabled processor. CUDA-enabled GPUs have hundreds of cores that can collectively run thousands of computing threads. Each core has shared resources, including registers and memory. The on-chip shared memory allows parallel tasks running on these cores to share data without sending it over the system memory bus .Thread hierarchy, shared memories and barrier synchronization are the three key abstractions of CUDA. A kernel can be executed by a one dimensional or two dimensional grids of multiple equally-shaped thread blocks. A thread block is a 3, 2 or 1-dimensional group of threads. Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. Threads in different blocks cannot cooperate and each block can execute in any order relative to other blocks. The number of threads per block is therefore restricted by the limited memory resources of a processor core.

CUDA kernel function is a fundamental building block of CUDA programs. When launching a CUDA kernel function, a developer specifies how many copies of it to run. We call each of these copies a task. Because of the hardware support of the GPU, each of these tasks can be small, and the developer can queue hundreds of thousands of them for execution at once. These tasks are organized in a two-level hierarchy, block and grid. Small sets of tightly coupled tasks are grouped into blocks. In a given execution of a CUDA kernel function, all blocks contain the same number of tasks. The tasks in a block run concurrently and can easily communicate with each other, which enables useful optimizations such as those of the section "Shared Memory". GPU's hardware keeps multiple blocks in flight at once, with no guarantees about their relative execution order. As a result, synchronization between blocks is difficult. The set of all blocks run during the execution of a CUDA kernel function is called a grid.

.



(Fig.1)

Threads in different blocks cannot cooperate and each block can execute in any order relative to other blocks. [11] The number of threads per block is therefore restricted by the limited memory resources of a processor core. On current GPUs, a thread block may contain up to 512 threads. The multiprocessor SIMT (Single Instruction Multiple Threads) unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.

3.4. GPU DATA TYPES

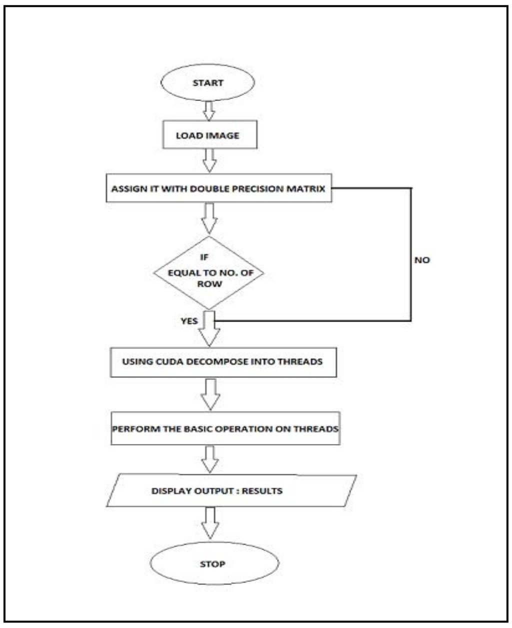
Jacket provides GPU counterparts to MATLAB's CPU data types, such as real and complex double, single, uint32, int32, logical, etc. Any variable residing in the host (CPU) memory can be cast to Jacket's GPU data types. Jacket's memory management system allocates and manages memory for these variables on the GPU automatically, behind the scenes. Any functions called on GPU data will execute on the GPU automatically without any extra programming, GPU function Jacket provides the largest available set of GPU functions in the world, ranging from functions like sum, sine, cosine, and complex arithmetic to more sophisticated functions like matrix inverse, singular value decomposition, Bessel functions, and Fast Fourier Transforms. The supported set of functions continues to grow with every release of Jacket (see the Function Reference Guide), runtime of jacket is the most advanced GPU runtime in the world, providing automated memory management, compile on the fly, and execution optimizations for Jacket enable code, Jacket's Graphics Toolbox is the only tool in the world that enables a merger of GPU visualizations with computation. With Jacket a simple graphics command can be added at the end of a simulation loop to visualize data as it is being computed while maintaining performance, The Developer SDK makes integration of custom CUDA code into Jacket's runtime very easy. With a few simple SDK functions, your CUDA code can benefit from the optimized Jacket platform. When Jacket applications have completed the development, test, and optimization stages and are ready for deployment, the Jacket MATLAB Compiler allows users to generate license free executables for distribution to larger user bases. (See the SDK and JMC Wiki pages) and Interactive help for any Jacket function is available using Jacket's ghelp function.

4. IMPLEMENTATION

In this section, first, we introduce the general scheme for Euclidean Distance and Mahalanobis Distance. Then, we introduce our GPU implementation environment by first discussing why GPUs are a good fit for medical imaging applications and then presenting NVIDIA's CUDA platform and GeForce 9500 GT architecture. Next, we talk about the CPU implementation environment. This is followed by description of the test data used in the experiments. Finally, we provide the list of CUDA kernels used in our GPU implementation.

4.1. FLOWCHART

This Flowchart helps to explain the process diagrammatically. First, take input as an image, read it as a matrix. Then decomposes the image into blocks, then from blocks to many columns. This process is repeated for all the rows. Using CUDA we decompose it in threads. Then the basic operation is performed.


START
LOAD IMAGE
ASSIGN IT WITH DOUBLE PRECISION MATRIX
IF EQUAL TO NO. OF ROW
NO
YES
USING CUDA DECOMPOSE INTO THREADS
PERFORM THE BASIC OPERATION ON THREADS
DISPLAY OUTPUT : RESULTS
STOP

(Fig. 2)

```
1. Load real Time RGB image.
2 Assign it with Double Precision Matrix.
3 Repeat for Each block.
4. Using CUDA decomposes into threads.
5. Perform the basic Operation (respective Distance formula)
6. Display the calculated.
7. Display the time require
```

Pseudo Code

4.3. CPU IMPLEMENTATION ENVIRONMENT

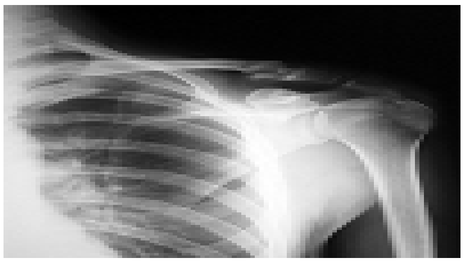
The CPU version of Distance formula is implemented in Matlab with integration of jacket v1.3.0. The computer used for the sequential implementation is an Intel® Dual-Core E5500 @ 2.80GHz and 2.0 GB of main memory which run on Windows XP. The algorithm was implemented in both single threading mode and multi-threading mode. Open MP is used to implement the multi-threading part.

4.4. GPU IMPLEMENTATION ENVIRONMENT

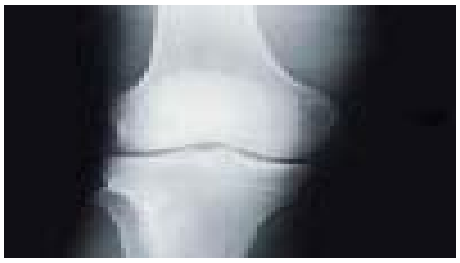
We have implemented Distance formula with NVIDIA’s GPU programming environment, CUDA v0.9. The era of single-threaded processor performance increases has come to an end. Programs will only increase in performance if they utilize parallelism. However, there are different kinds of parallelism. For instance, multi-core CPUs provide task-level parallelism. On the other hand, GPUs provide data-level parallelism. Depending on the application area, the type of the preferred parallelism might change. Hence, GPUs is good fit for all problems. However, medical imaging applications are very suitable to be implemented on GPU architecture. It is because these applications intrinsically have data-level parallelism with high compute requirements, and GPUs provide the best cost-per-performance parallel architecture for implementing such algorithms. In addition, most medical imaging applications (e.g. semi-automatic segmentation) require, or benefit from visual interaction and GPUs naturally provide this functionality. Hence, the use of the GPU in non-graphics related highly-parallel applications, such as medical imaging applications, became much easier than before. For to the graphics API. Since it is cumbersome to use graphics APIs for non-graphics tasks such as medical applications, instance, NVIDIA introduced CUDA to perform data-parallel computations on the GPU without the need of mapping the graphics-centric nature of previous environments. GPU- specific details and allowing the programmer to think in terms of memory and math operations as in CPU programs, instead of primitives, fragments, and textures that are specific to graphics programs [10]. CUDA is available for the NVIDIA GeForce 8400 (G80) Series and beyond. The GeForce 8400 GS model having 256Mbytes of video memory of DDR2 type and bus width of 64bit l. The memory bandwidth of the GeForce 8400 GTX is 80+ GB/s. To get the best performance from G80 architecture, we have to keep 128 processors occupied and hide memory latency. In order to achieve this goal, CUDA runs hundreds or thousands of fine, lightweight threads in parallel. In CUDA, programs are expressed as kernels. Kernels have a Single Program Multiple Data (SPMD) programming model, which is essentially Single Instruction Multiple Data (SIMD) programming model that allows limited divergence in execution. A part of the application that is executed many times, but independently on different elements of a dataset, can be isolated into a kernel that is executed on the GPU in the form of many different threads. Kernels run on a grid, which is an array of blocks; and each block is an array of threads. Blocks are mapped to multiprocessors within the G80 architecture, and each thread is mapped to single processor. Threads within a block can share memory on a multiprocessor. But two threads from two different blocks cannot cooperate. The GPU hardware performs switching of threads on multiprocessors to keep processors busy and hide memory latency. Thus, thousands of threads can be in flight at the same time, and CUDA kernels are executed on all elements of the dataset in parallel. We would like to mention that in our implementation; increasing the dataset size does not have an effect on the shared memory usage. This is because, to deal with larger datasets, we only increase the number of blocks and keep the shared memory allocations in a thread as well as the number of threads in a block the same.

5. RESULTS AND DISCUSSIONS

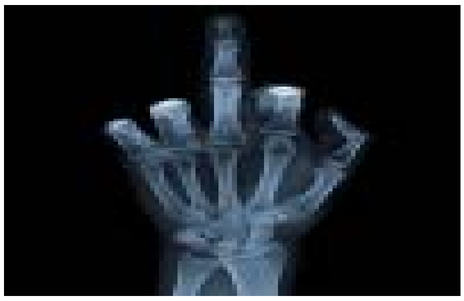
In this section, we first compare the runtimes of our GPU and CPU implementations for datasets with different sizes and present our speedup. Then, we show visual results by providing slices from one of the datasets. Next, we provide the breakdown of GPU implementation runtime to the CUDA kernel. Finally, we compare our implementation with Sharpetal’s implementation and highlight our improvements. We have achieved around 10X speedup over a single-threaded CPU implementation over GPU. We tested the codes by running them on CPU and GPU. Our test data contain the images shown in fig. 3-8.



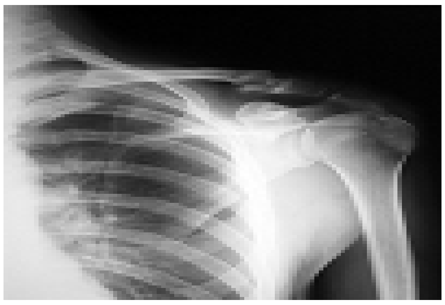
X-ray
(Fig.3)



X-ray
(Fig.4)



X-ray
(Fig.5)



X-ray
(Fig.6)

X-ray
(Fig. 7)

X-ray
(Fig. 8)

We got the following results for Euclidean Distance on CPU and GPU respectively:

Image and its Dimension	CPU-TIME (ms)	GPU-TIME (ms)
X-ray(32*32)	105.2	11.4
X-ray (100*100)	117	13.5
X-ray (500*500)	231.2	24.2
X-ray (800*800)	420.4	43.7

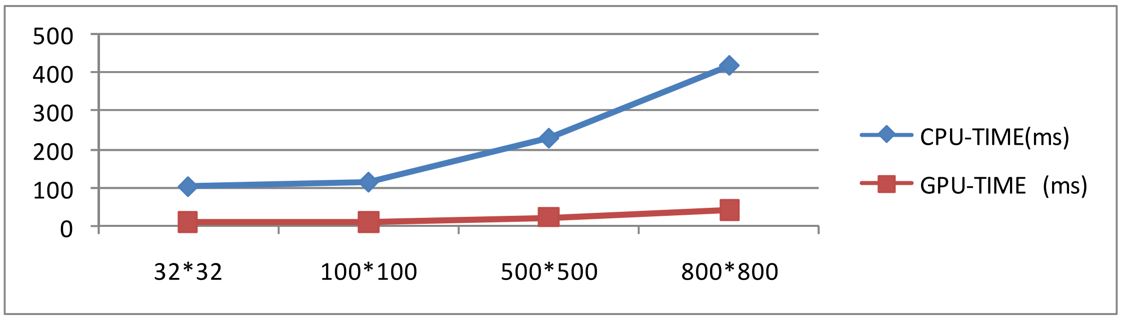
(Table 1)

The following results for Mahalanobis Distance on CPU and GPU respectively:

Image and its Dimension	CPU-TIME (ms)	GPU-TIME (ms)
X-ray (32*32)	145.4	15.4
X-ray (100*100)	152.2	15.9
X-ray (500*500)	243.2	25.7
X-ray(800*800)	364.2	37.2

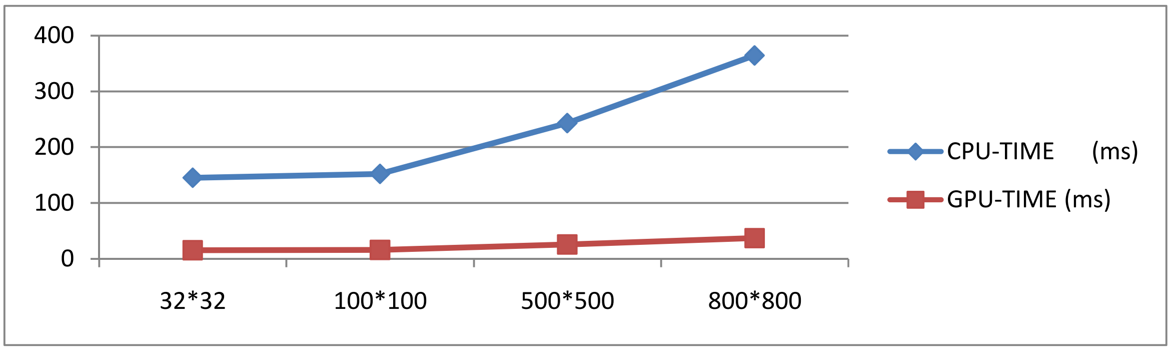
(Table 2)

6.1. COMPARITIVE GRAPH OF CPU AND GPU FOR EUCLIDEAN DISTANCE ALGORITHM



(Figure 9)

6.2. COMPARITIVE GRAPH OF CPU AND GPU FOR MAHALANOBIS DISTANCE ALGORITHM



(Figure 10)

CONCLUSION

The work presented in this paper shows the difference in the processing time of CPU and GPU when implemented on images in compressed domain. It shows the difference in time for each image at different sizes by implementing them in the Euclidean Distance and Mahalanobis Distance formula. We used a NVIDIA GeForce series 9500GT model having 1023 MB of video memory which reduced the process time from 150ms to 15ms. We obtained the speedup of the system 10 x times.

REFERENCES

[1] W. M. Smeulders, M. Worring, S. Santini, R. Jain and A. Gupta, “Content-Based Image Retrieval at the End of Early Years,” IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.22, No.12, 2000.
[2] J. P. Eakins and M. E. Graham, “A Report to the JISC Technology Applications Programme,” Institute for Image data Research, University of Northumbria at Newcastle, 1999.
[3] V. Castelli and L.D. Bergman (Editors), Image Databases: Search and Retrieval of Digital Imagery, J. Wiley &Sons, NY, 2002.
[4] Visual Communications and Image Processing, Touradj Ebrahimi, Thomas Sikora, Editors, Proceedings of SPIE Vol. 5150, 2003.
[5] Huiyu Zhou*, Abdul H. Sadka, Mohammad R. Swash, Jawid Azizi and Abubakar S. Umar : Content Based Image Retrieval and Clustering:A Brief Survey 2009 , Vol.1.
[6] Liwei Wang, Yan Zhang, Jufu Feng: On the Euclidean Distance of Images, 2005, vol.27, pg.-1334-1339.
[7] Sharadh Ramaswamy and Kenneth Rose: fast adaptive mahalanobis distance-based search and retrieval in image databases, Dec.2009, vol 18, pp. 1057-7149.
[8] T. Huang and X. S. Zhou, “Image retrieval with relevance feedback: From heuristic weight adjustment to optimal learning methods,” in ICIP, 2001, vol. 3, pp. 2–5.
[9] Kuldeep Yadav, Ankush Mittal M. A. Ansari , Avi Srivastava :Parallel Implementation of Compressed Sensing Algorithm on CUDA-GPU, IJCSIS-2011 , vol.9, No. 3 , pp. 112-119.
[10] NVIDIA CUDA Programming Guide,Version 2.2, page10,27-35,75-97,2009.
[11] Sanyam Mehta, Arindam Misra, Ayush Singhal, Praveen Kumar, Ankush Mittal, Kannappan Palaniappan: Parallel Implementation of Video Surveillance Algorithms on GPU Architecture using CUDA ,2009,