CHAPTER - 1

INTRODUCTION

"Image Processing Using CUDA" deals with performing some basic image operations such as image rotation and zooming on a parallel architecture such as a GPU i.e. Graphical Processing Unit using NVIDIA's CUDA i.e. Compute Unified Device Architecture.

1.1 Problem Definition

As these operations are highly computation intensive since it involves computations on every pixel of an image, it takes a lot of time to perform them serially for sufficiently large images. These operations are performed for each pixel of an image and all such pixels are evaluated in independent manner. This property can be utilized and these operations can be performed in parallel manner that is efficient to provide tremendous speed increases over CPU-only implementations and without any compromise in final image quality.

1.2 Aims and Objectives

- To develop software that will be able to perform image rotation, zooming and shrinking operations on both CPU and GPU for the images of high resolution.
- To study the serial algorithms for these operations and to implement them on CPU using C language.
- To develop equivalent fast and efficient parallel algorithms for these operations that will be implemented on the GPU using CUDA architecture utilizing all the resources and computing capabilities of the Graphics Processing Unit to obtain good results.
- To compare and study the time taken by these operations on the CPU and GPU and to accelerate them on GPU.

1.3 Feasibility Study

1. *Technical Feasibility*: The software has the following minimum requirements: A graphics card that supports CUDA 1.2 or later, CUDA SDK 1.2, Linux or Windows Operating System.

2. *Economical Feasibility*: The use of the graphics card increases the computational power of the system. Thus it reduces the cost of the system that supports scalar processors. It can well replace the computational power of a supercomputer.

3. *Operational Feasibility*: The use of the graphics card for general purpose programming has tremendously increased the computational power of the application. The average speed up that has been obtained has been around 30 times. This is an evidence of saving the time and resources of the application.

1.4 Preliminary Investigation

1.4.1 Image Processing

Image Processing deals with various operations that can be performed on images for enhancing their quality to study them. Some such operations are affine transformations (such as rotation, zooming, shrinking, translation and shear operations), image differencing, image registration, image segmentation, object recognition etc. Interest in digital image processing methods stems from two principal application areas:

i. improvement of pictorial information for human interpretation
ii. processing of image data for storage, transmission, and representation for autonomous machine perception

1.4.2 Image

An image is represented in the form of a two dimensional function *f(x, y)* where *x* and *y* are *spatial* (plane) coordinates and the amplitude of *f* at any pair of coordinated (*x*, *y*) is called the *intensity* or *gray level* of image at that point. When *x*, *y* and the amplitude values of *f* are all finite and discrete quantities, we call the image a *digital image*. A digital image is composed of a finite number of elements such that each have a particular

location and value. These elements are called *pixels* or *pels*. Images can either be gray scale images or colored images.

Fig 1.1 A gray scale image

Gray scale images can depict a small range of properties only as compared to color images. The use of color images is motivated by two principle factors:

i. Color is a powerful descriptor that often simplifies object identification.

ii. Human can discern thousands of color shades and intensities, compared to about only about two dozen shades of gray.

Color images are in the form of RGB format i.e. combination of colors Red, Green and Blue that are called primary colors of light. Each color in the RGB triplet is expressed as a value between 0-255.

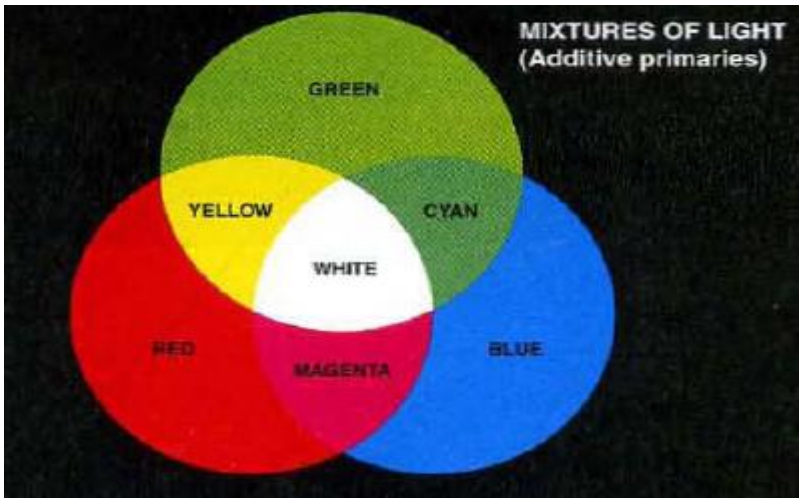


Fig 1.2 Primary colors of light (R, G, B)

1.4.3 ppm format of images

Images can be represented in many formats such as: ppm, jpeg, gif, bitmap. PPM format refers to *Portable Pixel Map* graphics. A ppm format image can be easily converted to other formats by using converters. The ppm format is the basic format of images in which the images are stored in form of RGB triplets such as (R, G, B) where each value varies in the range of 0-255. There are variations in ppm formats also that ranges from P1 to P6 formats.

P3 format images store the pixel positions and values in the form of a text file as shown below:

```
P3
# The P3 means colors are in ASCII, then 3 columns and 2 rows, then 255 for max color, then
RGB triplets
3 2
255
255  0  0        0 255  0        0  0 255
255 255  0       255 255 255      0  0  0
```

Fig 1.3 A sample (2 X 3) pixels ppm format image

The sample images of P3 format used in the project are:

Fig 1.4 blackbuck.ppm (512 X 512) pixels

Fig 1.5 snail.ppm (256 X 256) pixels

1.4.4 Image Rotation

Image rotation is an operation that is performed to change the orientation of an image by some specified angles. These angles can be defined arbitrarily. Rotation is about moving an image along the same axis and rotating it about some angle. Rotation is most commonly used to improve the visual appearance of an image, although it can be useful as a preprocessor in applications where directional operators are involved. In 2D the rotation is performed by multiplying the pixel positions by a matrix in terms of angle of rotation Θ for clockwise or counter-clockwise rotation about the origin. The resultant position defines the new pixel position of the rotated image. In most implementations, output locations which are outside the boundary of the image are ignored. There are many approaches to perform image rotation:

i. Affine Transformation

ii. Rotation by Shear Transformations

iii. Rotation by Area Mapping

1. Affine Transformation

It is a special case of projective transformations that preserves collinearity and ratio of distances. In affine transformation each pixel coordinate is multiplied with a transformation matrix that gives the new coordinates for that pixel. Thus rotation is performed in following steps:

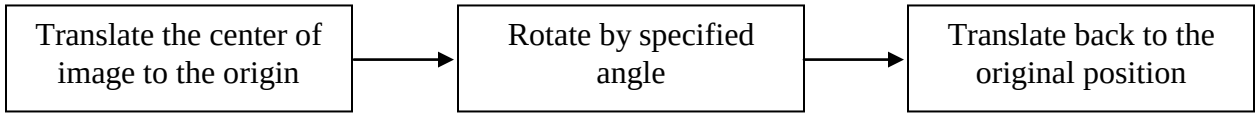
i. The creation of new pixel locations by multiplying them to the rotation transformation matrix.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

ii. The assignment of gray levels to those new locations same as those of old pixel positions.

The above transformation is applied as follows in order to rotate an image:

1. Translate the origin to centre of the image.
2. Rotate by specified angle in specified direction.
3. Translate the origin back to original position.



The above steps are depicted in the figure below in the order.

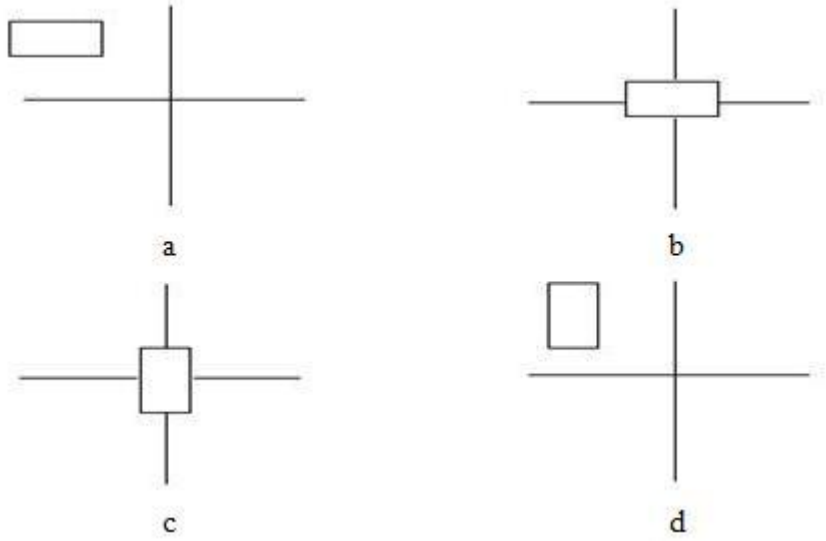


Fig 1.6 Rotation by affine transformation

2. Rotation by Shear Transformation

Rotation can be performed by horizontal and vertical shears. A horizontal shear moves a row of image pixels a horizontal distance that is proportional to its vertical distance from some reference point. Similarly a vertical shear moves a column of image pixels a vertical distance that is proportional to its horizontal

distance from some reference point. A succession of 3 alternating horizontal and vertical shears and be used to rotate an image perfectly by any angle. For rotation by small angles of 0.05 radians (about 3 degrees) or less, rotation can be performed by using 2 shears only.

Expressing in radians distortion introduced is:

$$\text{delta (length/width)} = 0.5 * \text{angle}^2$$

For such small angles this distortion is negligible. It's flexible, can be applied to pixels of any depth and is easily implemented as an in-place operation, changing the pixel values of the source image directly.

Let us take a sample image as:

a) Shear in x-direction and transpose rotates about the diagonal

b) Shear in y direction and transpose yields rotation bitmap

Fig 1.7 a) and b)

1.4.5 Image Zooming

Zooming and shrinking of an image deals with over-sampling and under-sampling of images. *Zooming* may be viewed as *oversampling*, while *shrinking* may be viewed as *under sampling*. Over-sampling is done by multiplying the number of pixels and copying the values into new pixels similar to their neighbor pixels. And under-sampling is done in just the opposite manner by decreasing the number of pixels and smoothing the image. The key difference between these two operations and sampling and quantizing an original continuous image is that zooming and shrinking are applied to a digital image unlike analog signals. Scaling is used to change the visual appearance of an image, to alter the quantity of information stored in a scene representation, or as a low-level preprocessor in multi-stage image processing chain which operates on features of a particular scale. Zooming requires *two steps*:

1. The creation of new pixel locations.
2. The assignment of gray levels to those new locations by interpolation such as *nearest neighbor interpolation* which can be accomplished by *pixel replication* and *bilinear interpolation.*

Duplication and *scaling* are done for zooming and *deletion* is done for shrinking. An image (or regions of an image) can be zoomed either through pixel replication or interpolation. Figure shows how pixel replication simply replaces each original image pixel by a group of pixels with the same value (where the group size is determined by the scaling factor). Alternatively, interpolation of the values of neighboring pixels in the original image can be performed in order to replace each pixel with an expanded group of pixels.

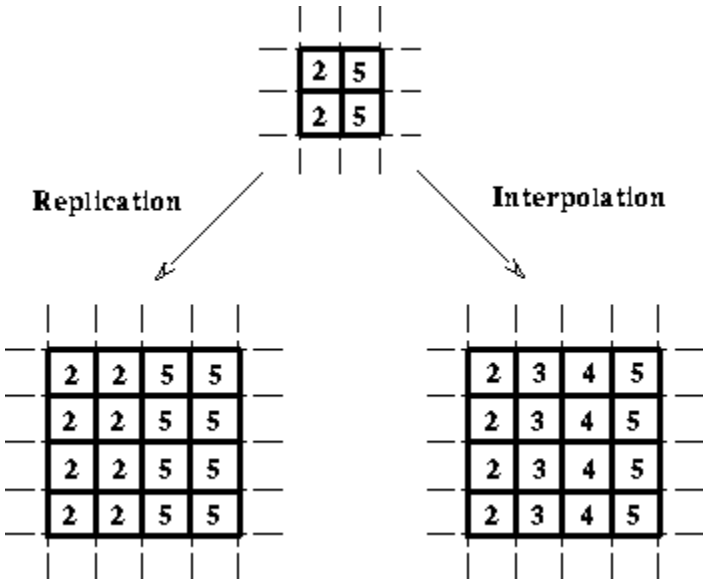


Fig 1.8 Image Zooming Methods

Replication simply involves copying the intensity values to new pixel locations.

Interpolation is the process of estimating the values of a continuous function from discrete samples. It provides a perfect reconstruction of a continuous function, provided that the data was obtained by uniform sampling at or above Nyquist rate (a sampling rate equal to exactly twice the highest frequency). The general form of interpolating function is:

$$g(x) = £_k\ c_k\ u(\text{distance}_k)$$

where g() is the interpolation function, u() is the interpolation kernel, c_k is the interpolating coefficient, and distance of x from x_k.

Interpolation methods can be used for zooming such as:

1. Nearest neighbor method
2. Bilinear interpolation
3. Bicubic interpolation
4. Pixel art scaling algorithms

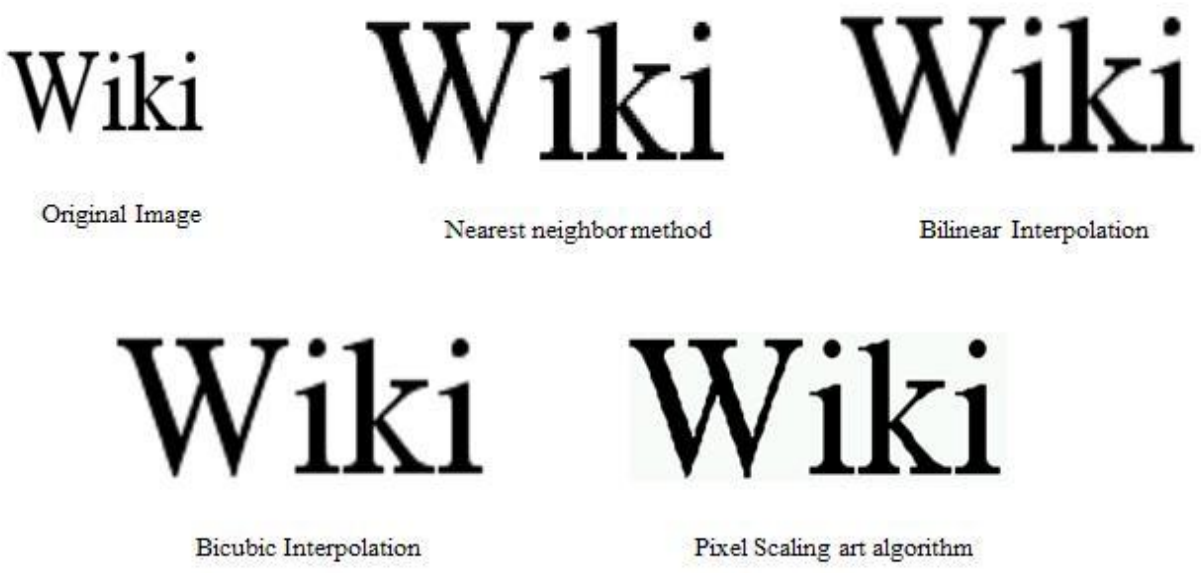


Fig 1.9 Interpolation Methods

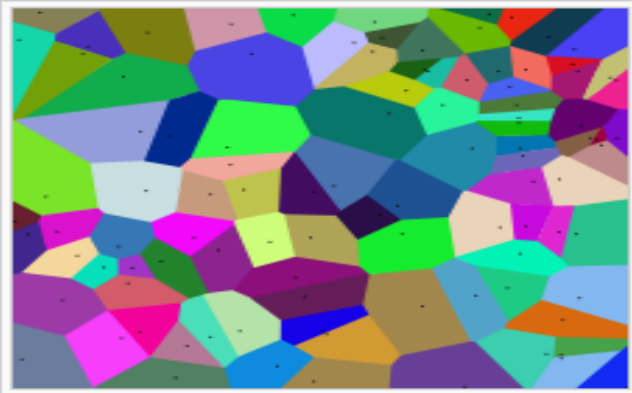
1. Nearest Neighbor Method

The nearest neighbor algorithm simply selects the value of the nearest point, and does not consider the values of other neighboring pixel at all, yielding a piecewise-constant interpolant. It is also known as proximal interpolation or point sampling. This method does not actually interpolate values but it just copies the values. It is used in real-time 3D rendering to select color values for a textured surface. Although nearest neighbor interpolation is fast, it has the undesirable feature that it produces a checkerboard effect that is objectionable at high level of magnification. Aliasing effects can be removed by *blurring* the image before zooming it.

For 1D number of grid points needed to evaluate interpolation function are 2. For 2D number of grid points needed to evaluate interpolation function are 4. The interpolation kernel for each direction is :

$$u(s) = \begin{cases} 0 & |s| > 0.5 \\ 1 & |s| < 0.5 \end{cases}$$

and $c_k = f(x_k)$.



Example of nearest neighbor interpolation of a random set of points (black dots) in 2D. Each coloured cell indicates the area in which all the points have the black point in the cell as their nearest black point.

Fig 1.10 Nearest Neighbour Method

2. Bilinear Interpolation

This method determines the gray level value from the weighted average of the four closest pixels to the specified input coordinates, and assigns that value to the output coordinates. One linear interpolation is performed in horizontal direction and again one linear interpolation is performed in its perpendicular direction i.e., vertically. For 1D, number of grid points needed is 2 and for 2D number of grid points needed are 4.

$$u(s) = \begin{cases} 0 & |s| > 1 \\ 1 - |s| & |s| < 1 \end{cases}$$

Suppose that we want to find the value of the function f at that point P=(x,y). It is assumed that we know the value of f at four points $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$.

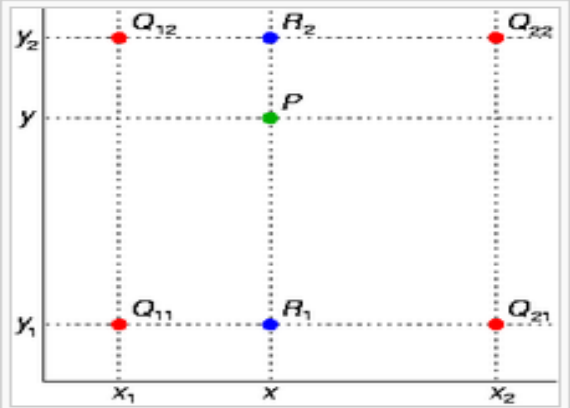


Fig 1.11 Bilinear Interpolation

We first do linear interpolation in the *x*-direction. This yield

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

where $R_1 = (x, y_1)$,

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

where $R_2 = (x, y_2)$.

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2).$$

We proceed by interpolating in the *y*-direction.

$$
\begin{aligned}
f(x, y) \approx & \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y) \\
& + \frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y_2 - y) \\
& + \frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y - y_1) \\
& + \frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y - y_1).
\end{aligned}
$$

This gives us the desired estimate of $f(x, y)$.

Contrary to what the name suggests, the bilinear interpolant is not linear; rather, it is a product of two linear functions: $(a_1x + a_2)(a_3y + a_4)$

Alternatively, the interpolant can be written as: $(b_1 + b_2x + b_3y + b_4xy)$.

3. Bicubic Interpolation

It determines the gray level value from the weighted average of 16 closest pixels to the specified input coordinates, and assigns that value to the output coordinates. The image is slightly sharper than that produced by Bilinear Interpolation and it does not have disjointed appearance produced by Nearest Neighbor Interpolation. For 1D, the number of grid points needed is 4, two grid points on either side of the point under consideration. For 2D, the number of grid points needed is 16, two grid points on either side of the point under consideration for both horizontal and vertical directions. For 1D interpolation kernel is:

$$
u(s) = \begin{cases} 3/2|s|3 - 5/2|s|2 + 1 & 0 <= |s| < 1 \\ -1/2|s|3 + 5/2|s|2 - 4|s| + 2 & 1 <= |s| < 2 \\ 0 & 2 < |s| \end{cases}
$$

For 2D interpolation, 1D interpolation function is applied in both directions. It's a separable extension of the dimensional interpolation function:

$$g(x,y) = \sum_{l=-1}^{2}\sum_{m=-1}^{2} c_{j+l,k+m}\, u(dis\tan ce_x)\, u(dis\tan ce_y)$$

$c_{jk} = f(x_j, y_k)$.

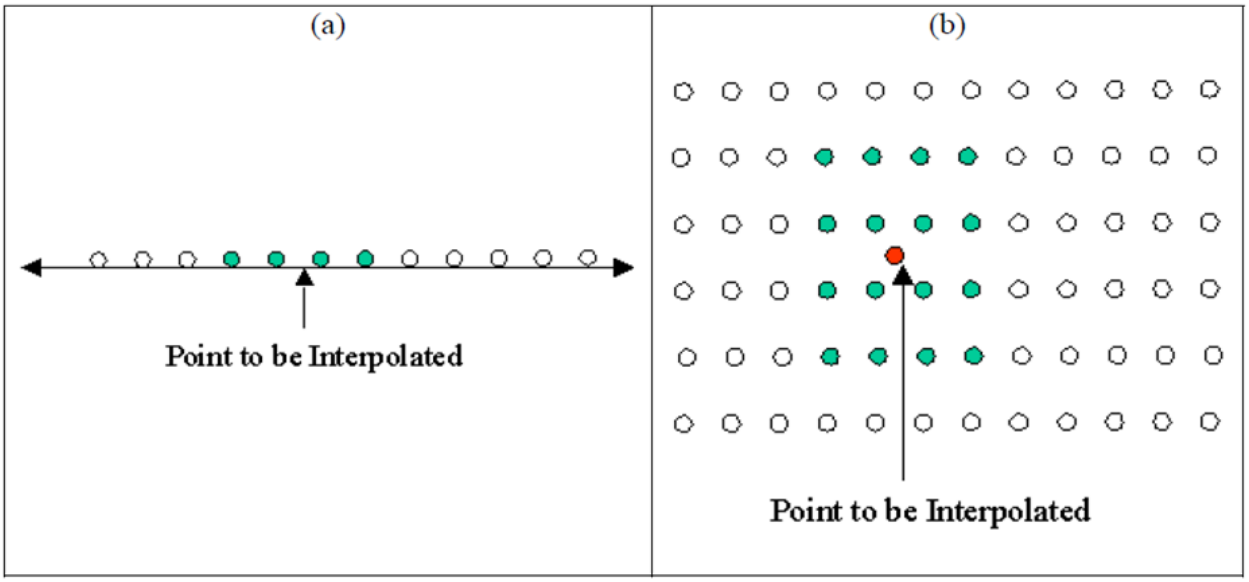


Fig 1.12 Grid points in (a)1D and (b)2D in cubic convolution interpolation

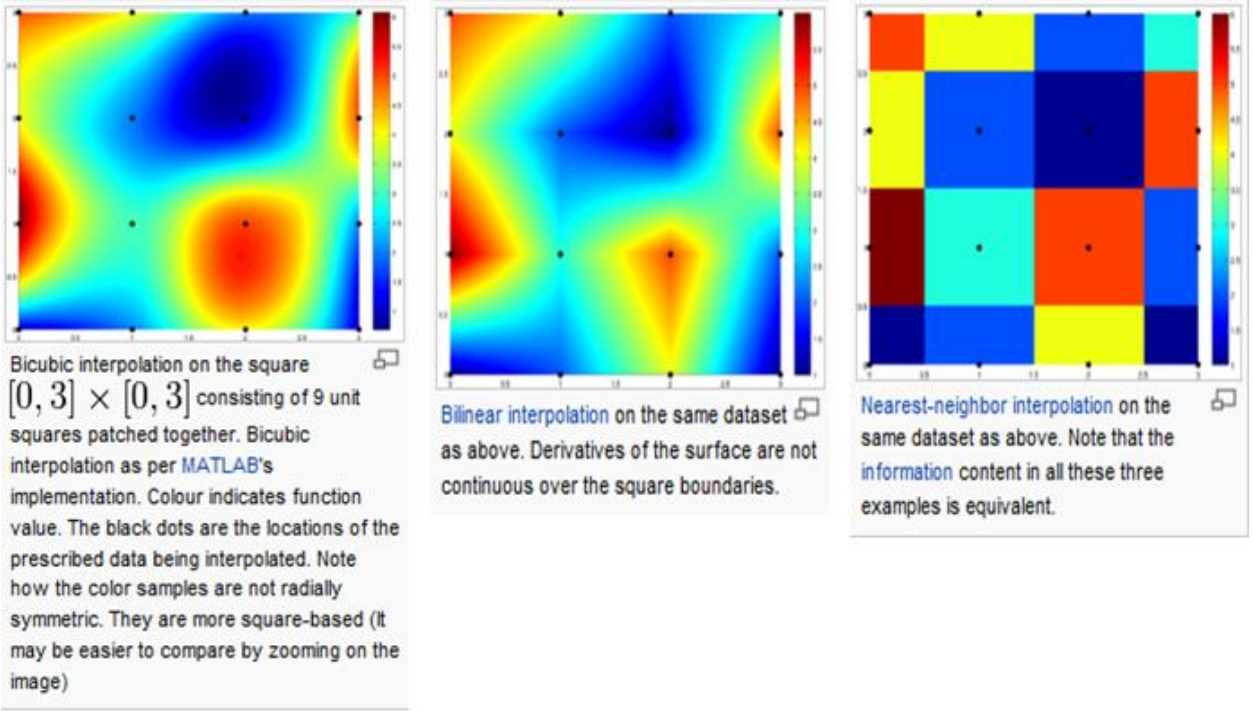


Fig 1.13 Comparison of nearest neighbour, bilinear and bicubic interpolation

1.4.6 CUDA

CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA. CUDA is an architecture used for performing computations in graphics processing units (GPUs). It is a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language.

CUDA C programming involves running the code on both the platforms concurrently i.e. a *host system* with one or more CPUs and one or more *devices* with CUDA enabled NVIDIA GPUs. It involves following tasks to be performed:

- *Parallel Computing with CUDA*: It deals with the important aspects of the parallel programming architecture.
- *Performance Metrics*: It provides a way to measure performance in CUDA applications and study the factors that most influence performance.
- *Memory Optimizations*: Correct memory management is one of the most effective means of improving performance. Different kinds of memory available to CUDA applications.
- *Execution Configuration Optimizations*: To make sure your CUDA application is exploiting all the available resources on the GPU.
- *Instruction Optimizations*: Certain operations run faster than others. Using faster operations and avoiding slower ones often confers remarkable benefits.
- *Control Flow*: Carelessly designed control flow can force parallel code into serial execution; whereas thoughtfully designed control flow can help the hardware perform the maximum amount of work per clock cycle.
- *Getting the Right Answer*: How to debug code and how to handle differences in how the CPU and GPU represent floating-point values.

The two primary **differences between host and device** are in threading and memory access.

- *Threading resources*. Execution pipelines on host systems can support a limited number of concurrent threads. Servers that have four quad-core processors today can run only 16 threads concurrently. By comparison, the smallest executable unit of parallelism on a CUDA device comprises 32 threads (a warp).

- *Threads*. Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off of CPU execution channels to provide multithreading capability. Context switches (when two threads are swapped) are therefore slow and expensive. By comparison, threads on GPUs are extremely lightweight. Because separate registers are allocated to all active threads, no swapping of registers or state need occur between GPU threads.

- *RAM*. Both the host system and the device have RAM. On the host system, RAM is generally equally accessible to all code (within the limitations enforced by the operating system). On the device, RAM is divided virtually and physically into different types, each of which has a special purpose and fulfills different needs.

The amount of **performance benefit** an application will realize by running on CUDA depends entirely on the extent to which it can be parallelized. *Amdahl's law* specifies the maximum speed-up that can be expected by parallelizing portions of a serial program. Essentially, it states that the maximum speed-up (S) of a program is

$$S = 1/((1-P) + P/N)$$

where P is the fraction of the total serial execution time taken by the portion of code that can be parallelized and N is the number of processors over which the parallel portion of the code runs.

The host **runtime component of the CUDA** software environment can be used only by host functions. It provides functions to handle the following:

- Device management
- Context management
- Memory management
- Code module management
- Execution control

- Texture reference management
- Interoperability with OpenGL and Direct3D

It comprises two APIs:

- A low-level API called the CUDA driver API
- A higher-level API called the C runtime for CUDA that is implemented on top of the CUDA driver API

There are basically five types of memories on a GPU:

1. *Global Memory*: It is present on DRAM off-chip
2. *Local Memory*: It is also present off-chip and access to it is as expensive as to access global memory. Its scope is local to a thread and is used only to hold local variables.
3. *Shared Memory*: It is present on-chip and is much faster than global and local memories. But it may suffer from the problem of bank-conflicts.
4. *Constant Memory*: There is a total of 64KB of constant memory. The constant memory space is cached. As a result, a read from constant memory costs one memory read from device memory only on a cache miss; otherwise, it just costs one read from the constant cache.
5. *Texture Memory*: It is a read only memory and is cached and thus faster than global memory.

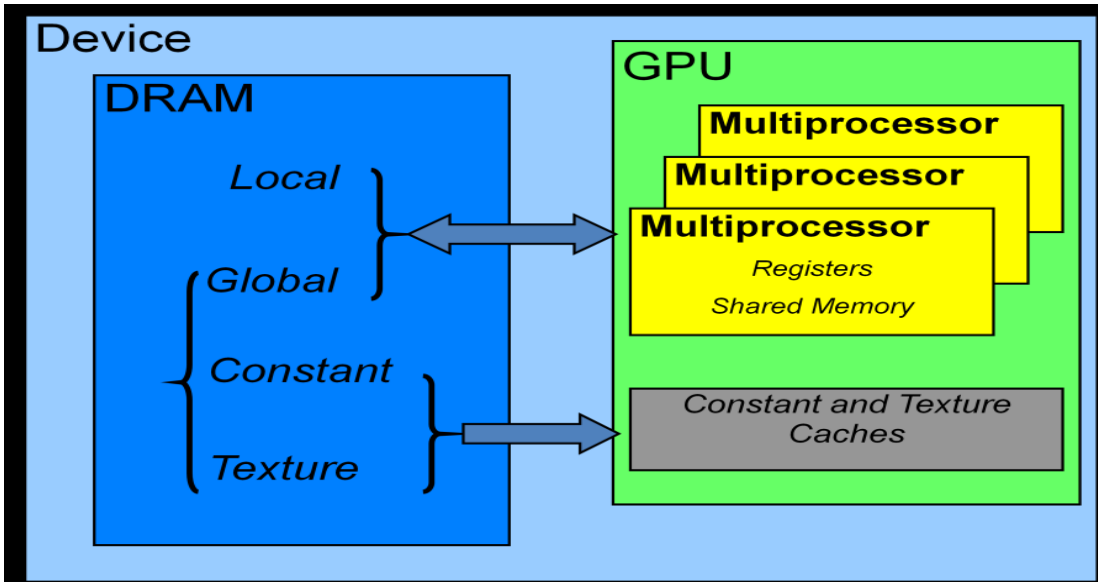


Fig 1.14 Memories on a GPU

Registers: It is present on-chip and accessing a register consumes zero extra clock cycles per instruction, but delays may occur due to register read-after-write dependencies and register memory bank conflicts.

A common problem that arises is **branch divergence.** Any flow control instruction (if, switch, do, for, while) can significantly affect the instruction throughput by causing threads of the same warp to diverge; that is, to follow different execution paths. If this happens, the different execution paths must be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path. This degrades the performance. To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps.

CHAPTER - 2
LITERATURE REVIEW

Driven by the insatiable market demand for real time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth.

In case of an image, we know that an image is made up of small units named as pixels. These pixels hold the information about an image. While doing operations like rotation or zooming of an image, all the pixels act independent of each other. The calculated value of each pixel in the resultant image is independent of the values of other pixels.

The existing algorithms are serial in nature and do not produce much efficiency. Especially if we consider an image of very large size, the execution times are considerably high. This aspect is highly unfavorable in real time applications.

When executing on a CPU, all the operations are performed in a serial manner, i.e. the order of execution is serial.

Exploiting this independency, if we can devise a mechanism where all the independent calculations can be done in parallel, we can certainly reduce the execution time.

So here comes the need of parallel programming and CUDA is such an architecture which describes a mechanism where the graphics card can produce independency in all the calculations. This differentiation between the CPU and the GPU becomes more and more evident as the size of the image grows. Thus for images of large size, image operations can be executed more efficiently and in less time on a GPU as compared to CPU. Here we have implemented this technique and devised a mechanism to execute this.

According to CHEN Xuede, LU Siwei, YUAN Xiaobu's "Midpoint Line Algorithm for High-speed High-accuracy Rotation of Images" in 1996, a one-pass method for image rotation has been devised. The method is straightforward, and usually

obtains higher accuracy than the multipass methods. However, the method is inefficient since it requires a large number of computations (the four floating-point multiplications and two floating-point additions for each pixel). It takes a very long time of actual calculation to rotate a large image with a lot of pixels.

J. Hinks, S.A. Amin says in "PARALLEL IMPLEMENTATION FOR IMAGE ROTATION USING PARALLEL VIRTUAL MACHINE (PVM)" that the inverse rotation transformation matrix will produce an image that'll have some pixels with more than one value that are overwritten and some pixels having no values which are left empty. This happens due to rounding off the floating values to nearest integer values.

NUI0 TSUCHIDA, YOJI YAMADA and MINORU UEDA's "Hardware for Image Rotation by Twice Skew Transformations" depicts that large coordinate errors may be accumulated by continuous additions, especially in the case of a small scale 8 bit microcomputer. Furthermore, it is difficult to produce hardware for image rotation based on such an ordinary method.

So the major problem highlighted is that the implementation of the basic algorithm for rotation of an image in large scale images is not very hardware efficient. Hence a separate mechanism has to be employed in order to execute these algorithms.

Considering Chun-Ho Kim, Si-Mun Seong, Jin-Aeon Lee, and Lee-Sup Kim's "Winscale: An Image-Scaling Algorithm Using an Area Pixel Model", it tells us an algorithm that is efficient in image scaling. But the algorithm takes much more time as the size of images increases.

Wen Sun, Yan Lu, Feng Wu, Shipeng Li's "REAL-TIME SCREEN IMAGE SCALING AND ITS GPU ACCELERATION" devises a method that is very efficient and is hardware accelerated, but has involved very complex operations such as POT Map calculation, WF Map calculation and Adaptive Post Processing.

So a new approach has to be developed that can use these basic algorithms for rotation and zooming of an image that are hardware efficient as well as less complex. So the introduction of parallel processing has been a need. The deficiencies that existed in hardware support have been removed through parallel processing. CUDA can act as a tool that can make these algorithms very powerful in the real world.

CHAPTER - 3

PROPOSED METHODOLOGY

Image rotation and image zooming operations as described can be implemented using C language on a CPU in a serial fashion by implementing the algorithms and methods defined. For images of considerably large sizes these operations takes a very long computation time when performed serially. The idea is to implement these operations in a parallel fashion such that all the pixels perform the parallel computation thus reducing the time of the operation.

3.1 Parallel Image Rotation

The serial way to implement image rotation is to implement the method of Affine Transformation in C on CPU. For parallel architecture the method of Affine Transformation can be modified to be implemented in parallel on a GPU. For each pixel launch as a single thread on GPU that do the following:

1. Translate the origin to centre of the image.
2. Rotate by specified angle in specified direction.
3. Translate the origin back to original position.

Threads are grouped in blocks such that multiple blocks run concurrently such that:

1. Maximum number of threads in a block is 512 and on a multiprocessor 768.
2. Maximum number of blocks on one multiprocessor is 8 and blocks are organized in grids.

The rotation operation produces output locations which do not fit within the boundaries of the image (as defined by the dimensions of the original input image). In such cases, destination elements which have been mapped outside the image are ignored. Pixel locations out of which an image has been rotated are usually filled in with black pixels as the background pixels.

The image obtained after rotation by above method has black zig-zag patterns that are of the kind of *Jaggies*. In order to generate the intensity of the pixels at each integer

position, different heuristics (or re-sampling techniques) may be employed. For example, two common methods include:

- Allow the intensity level at each integer pixel position to assume the value of the nearest non-integer neighbor pixel.
- Calculate the intensity level at each integer pixel position based on a weighted average of the n nearest non-integer values. The weighting is proportional to the distance or pixel overlap of the nearby projections.

The method that we have employed to remove these spots is by *smoothing* the image by using an *averaging filter* that re-computes the value of each pixel by finding average of its four horizontal and vertical neighbors.

The parallel version of the above algorithm is described here.

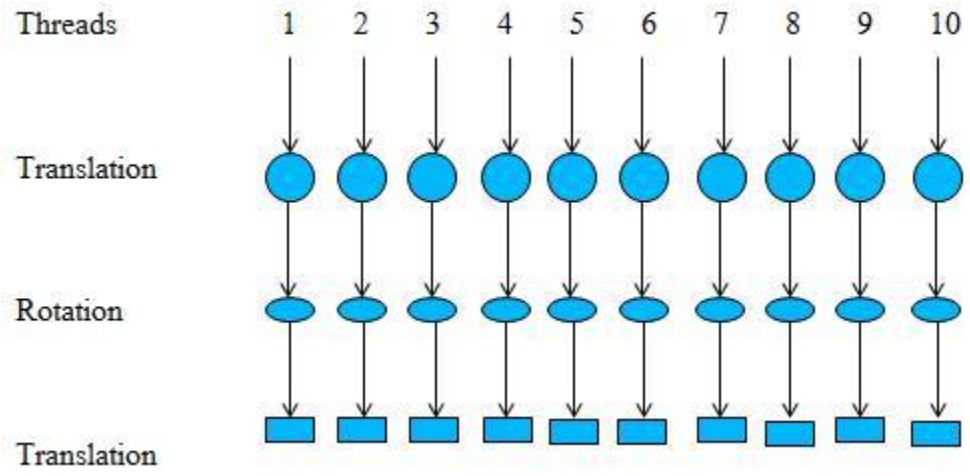


Fig 3.1 Parallel Rotation

The threads are organized in blocks and all the blocks are executed on the cores of the GPU using global memory. All threads of a single block execute concurrently. Thus the output image is obtained in array b[m][3n]. The quality of the image is further enhanced by applying averaging filter on the output image obtained.

3.2 Parallel Image Zooming

Image zooming operation is performed by bilinear interpolation. The method is implemented in C on CPU. The method of bilinear interpolation smoothens the image.

This method can be implemented in parallel by performing the operations for all pixels concurrently. For each pixel launch as a single thread on GPU that do the following:

1. Form the new grid and copy the values form original grid to new grid.
2. Perform horizontal interpolation.
3. Perform vertical interpolation.

The actual image is mapped on the new grid of the image formed by analysing the scaling factor of the image. This is depicted as follows:

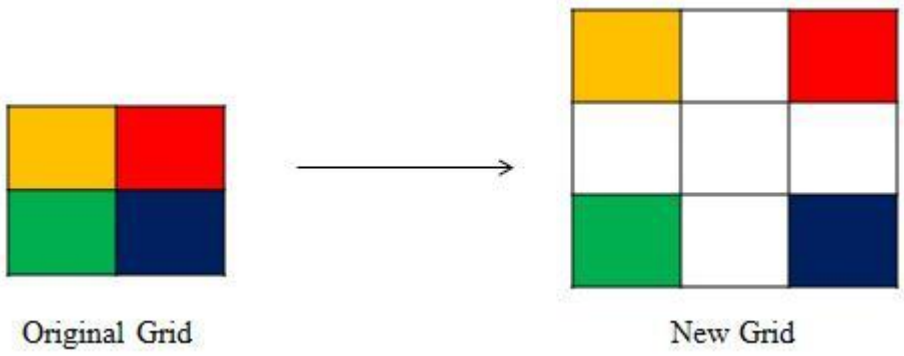


Fig 3.2 Copy pixel values from original to new grid.

The parallel algorithm is described below:

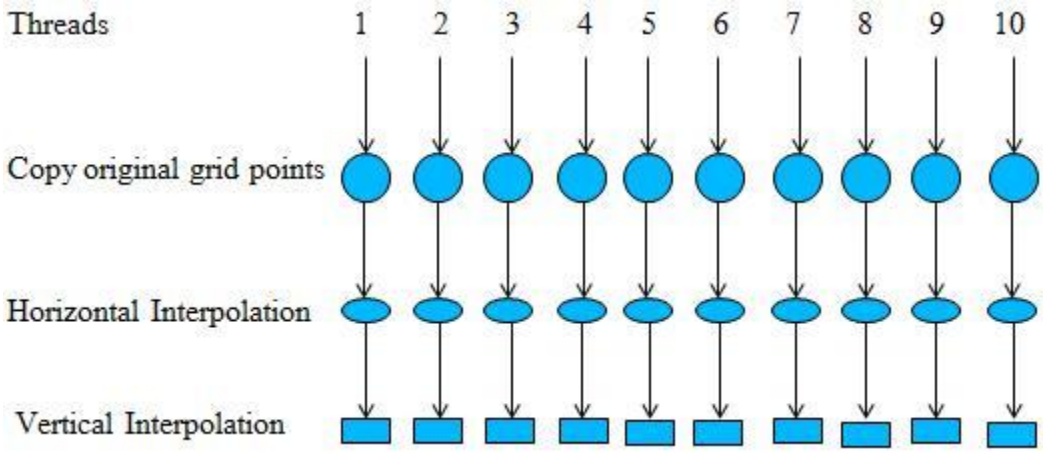


Fig 3.3 Parallel Zooming

The threads are organized in blocks and all the blocks are executed on the cores of the GPU using global memory. All threads of a single block execute concurrently. Thus the output image is obtained in array b[p][3q].

CHAPTER - 4

REQUIREMENT SPECIFICATION

The CUDA architecture can be implemented on a GPU using a NVIDIA's graphic card. The requirements for the software and hardware for our project is discussed in following sections.

4.1 Software Requirements

The project is equally supported by both the operating systems i.e. Windows XP, Windows Vista and Windows 7 along with Linux operating system particularly Fedora 10. CUDA SDK 1.2 or later is used for implementing CUDA APIs. The serial code can be implemented in C language using any editor for C. The parallel code is implemented using CUDA with its APIs. The NVIDIA graphic card drivers are used to enable the features of the graphic card.

4.2 Hardware Requirements

A graphics card that supports CUDA 1.2 or later can be used for the project.

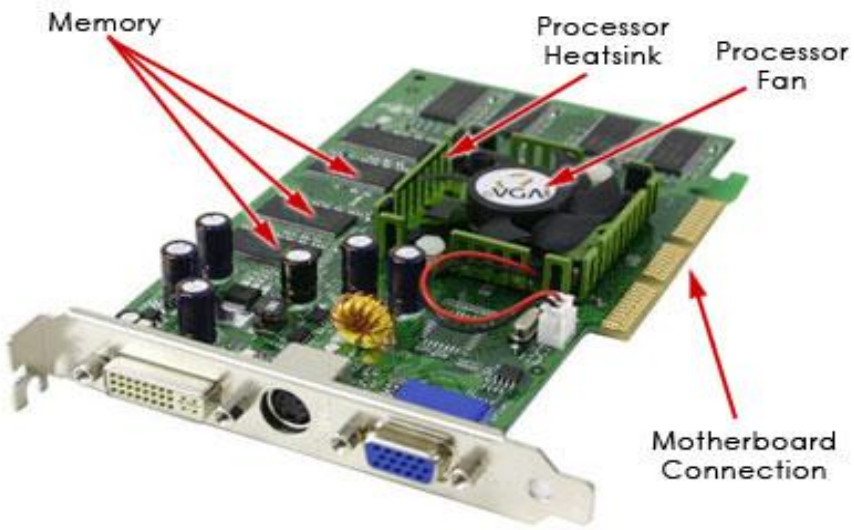


Fig 4.1 A Graphic Card

A graphics card, also known as a video card or graphics accelerator card, is a piece of computer hardware that is responsible for creating the images one can see on the monitor. It is usually used to refer to the type of card that is an expansion to the computer's motherboard, and not the one integrated into the computer already. They have been used in many other types of electronic devices, particularly game consoles.

A basic modern video card consists of several circuitry items.

- A Graphics Processing Unit (GPU) (more on GPU in subsequent lectures)
- Video Memory and BIOS
- A Random Access Memory Digital to Analog Converter (RAMDAC)
- Outputs and a cooling device.

A **Graphics Processing Unit** (GPU) is responsible for what one can see on the screen. It is a separate microprocessor that frees up the CPU. Unlike the CPU, the GPU makes use of much more complicated mathematical and geometrical equations to complete the graphics rendering. In our project we have used a 192 core GPU.

Video RAM or VRAM is the number of Megabytes one can see on the box of a graphics card (Up to 4GB DDR2 in NVIDIA). Graphics cards have their own memory space to help the computer process images. This part of the card makes use of the z-buffer. This is particularly helpful in 3D graphics, as it coordinates depth in an image. It is this memory that makes a 3D environment possible.

The **BIOS** is just the basic programming for interaction between the card and the computer. It contains the memory for all of the cycles the card must go through. It is an extremely important area on the card, and if damaged, will cause the card to stop working.

The **input/output ports** are the links from the video card to other instruments and peripherals. Common outputs are:

- SVGA outputs to the CRT monitors
- DVI outputs to digital outputs such as LCD monitors and projectors
- S-video outputs to TVs, Game consoles, and video recorders

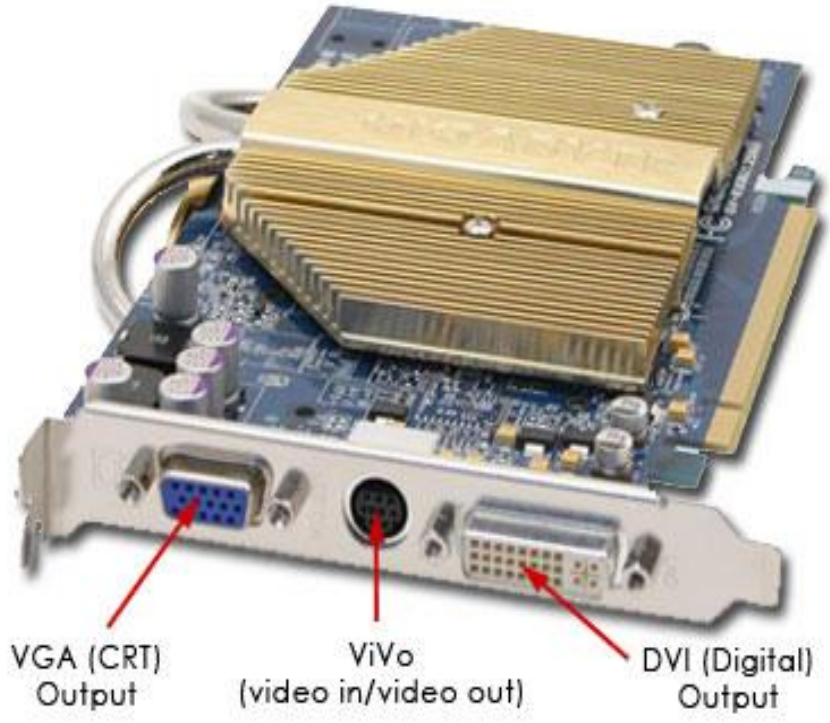


Fig 4.2 Input/output ports

Graphics cards connect to the computer through the motherboard. This supplies power to the card and lets it communicate with the CPU. Newer graphics cards often require more power than the motherboard can provide, so they also have a direct connection to the computer's power supply. *Connections to the motherboard* are usually through one of three interfaces:

1. Peripheral component interconnect (PCI)
2. Advanced graphics port (AGP)
3. PCI Express (PCIe)

PCI Express is the newest of the three and provides the fastest transfer rates between the graphics card and the motherboard. *PCIe also supports the use of two graphics cards in the same computer*. Most graphics cards have two monitor connections. Often, one is a DVI connector, which supports LCD screens, and the other is a VGA connector, which supports CRT screens. In addition to connections for the motherboard and monitor, some graphics cards have connections for:

- TV display: TV-out or S-video
- Analog video cameras: ViVo or video in/video out
- Digital cameras: FireWire or USB

PCI Express is a serial connection that operates more like a network than a bus. Instead of one bus that handles data from multiple sources, PCIe has a switch that controls several point-to-point serial connections. These connections fan out from the switch, leading directly to the devices where the data needs to go. Every device has its own dedicated connection, so devices no longer share bandwidth like they do on a normal bus.

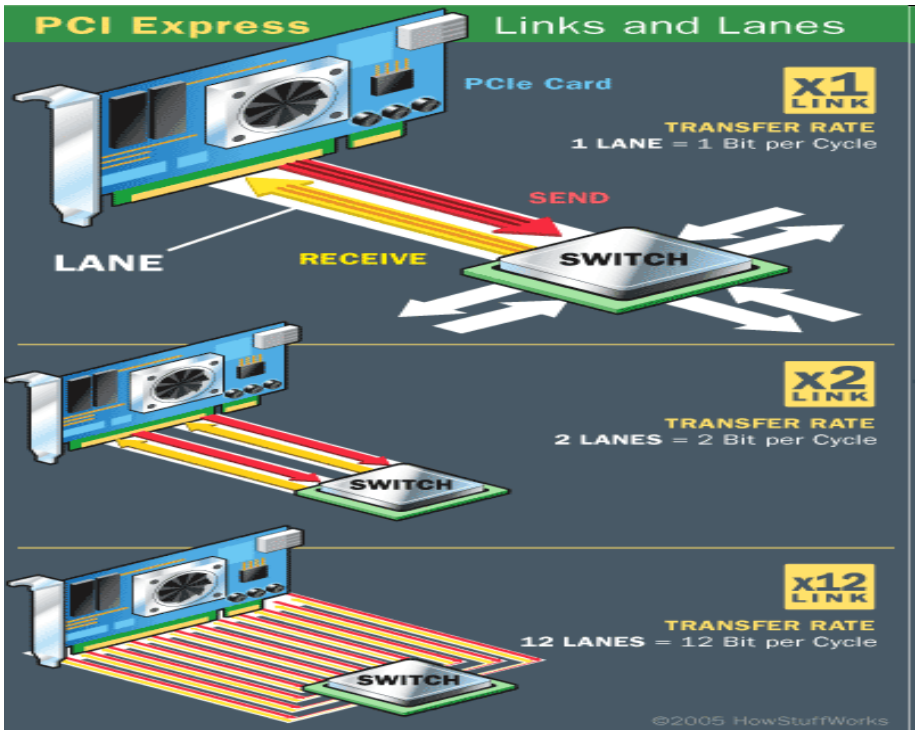


Fig 4.3 PCI Express

PCI Express uses 8b/10b encoding, which encodes 8-bit data bytes into 10 bit-transmission characters. This improves the physical signal; easier bit synchronizations, receivers and transmitters and simpler in design, error detection and correction is improved.

CHAPTER - 5

IMPLEMENTATION

The operations of image rotation and image zooming are implemented on both CPU using C and GPU using CUDA. The algorithms used for implementation are described as follows:

5.1 Image Rotation

The rotation of images is implemented serially on CPU by a serial algorithm and in parallel on GPU defined by the parallel algorithm. The serial and parallel algorithms are described below:

Serial Algorithm:

Input: an input image of m X n pixels is stored in an array a[m][3n] as there are three values i.e. Red, Green and Blue for each pixel. Θ be the angle of rotation.

Output: The output image will be stored as an array b[m][3n].

Variables: Center of rotation is denoted by (x_0, y_0). Pixels of input image are denoted by (x_1, y_1) and that of output image by (x_2, y_2).

serial_rotation (a, b, m, n, Θ)

{

for $x_1 = 0$ to m do

for $y_1 = 0$ to 3n do

$x_2 = \cos\Theta * (x_1-x_0) - \sin\Theta * (y_1-y_0) + x_0$

$y_2 = \sin\Theta * (x_1-x_0) + \cos\Theta * (y_1-y_0) + y_0$

$b[x_2][y_2] = a[x_1][y_1]$

end for

end for

}

Parallel Algorithm:

Input: an input image of m X n pixels is stored in an array a[m][3n] as there are three values i.e. Red, Green and Blue for each pixel. Θ be the angle of rotation.

Output: The output image will be stored as an array b[m][3n].

Variables: Center of rotation is denoted by (x_0, y_0). Pixels of input image are denoted by (x_1, y_1) and that of output image by (x_2, y_2).

Thread-Block Organization: Thread block is organized in two-dimensions with 256 threads per block.

parallel_rotation (a, b, m, n, Θ)

{

for each thread in a block such that each thread perform computation for one pixel

do in parallel

$x_2 = \cos\Theta * (x_1-x_0) - \sin\Theta * (y_1-y_0) + x_0$

$y_2 = \sin\Theta * (x_1-x_0) + \cos\Theta * (y_1-y_0) + y_0$

$b[x_2][y_2] = a[x_1][y_1]$

end for in parallel

}

5.2 Image Zooming

The zooming of images is implemented serially on CPU by a serial algorithm and in parallel on GPU defined by the parallel algorithm. The serial and parallel algorithms are described below:

Serial Algorithm:

Input: an input image of m X n pixels is stored in an array a[m][3n] as there are three values i.e. Red, Green and Blue for each pixel. α is the scaling factor.

Output: The output image will be stored as an array b[p][3q] where scaled image is of dimensions p X q.

Variables: Pixels of input image are denoted by (x_1, y_1) and that of output image by (x_2, y_2).

serial_zooming (a, b, m, n, α)

{

$p = \alpha*m - (\alpha-1)$

q = α*n - (α-1)
for x_1 = 0 to m do
for y_1 = 0 to 3n do
map the gird points in b[x_2][y_2] from a[x_1][y_1]
end for
end for
perform horizontal interpolation to obtain remaining grid points on horizontal line
perform vertical interpolation to obtain remaining grid points on vertical line
write the pixels of output image from b[p][3q]
}

Parallel Algorithm:

Input: an input image of m X n pixels is stored in an array a[m][3n] as there are three values i.e. Red, Green and Blue for each pixel. α is the scaling factor.

Output: The output image will be stored as an array b[p][3q] where scaled image is of dimensions p X q.

Variables: Pixels of input image are denoted by (x_1, y_1) and that of output image by (x_2, y_2).

Thread-Block Organization: Thread block is organized in two-dimensions with 256 threads per block.

parallel_zooming (a, b, m, n, α)
{
for each thread in a block such that each thread perform computation for one pixel
do in parallel
p = α*m - (α-1)
q = α*n - (α-1)
for x_1 = 0 to m do
for y_1 = 0 to 3n do
map the gird points in b[x_2][y_2] from a[x_1][y_1]
end in parallel

perform horizontal interpolation in parallel to obtain remaining grid points on horizontal line

perform vertical interpolation in parallel to obtain remaining grid points on vertical line

write the pixels of output image from b[p][3q]

}

CHAPTER – 6

TESTING

6.1 Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects). There are many methods to apply testing. Software testing is used in association with verification and validation:

Verification: Have we built the software right? (i.e., does it match the specification).

Validation: Have we built the right software? (i.e., is this what the customer wants).

6.2 Testing in our project

The code for image rotation and zooming are tested by executing them for different input values. The following test values are obtained by performing these operations for the image shown below:

Fig 6.1 blackbuck.ppm (512 X 512) pixels

For image rotation the computation time for different values of angle of rotation as obtained for above image are as follows:

Table 6.1 Rotation test data for different angles

Rotation Angle (degrees)	Processing Time on CPU (ms)	Processing Time on 192 core GPU(ms)	Speed up obtained
-180	10	0.26	38.46
-90	10	0.81	12.35
-45	20	0.5	40
0	10	0.23	43.47
15	10	0.31	32.26
30	20	0.42	47.62
45	10	0.55	18.18
60	10	0.63	15.87
75	10	0.72	13.89

90	10	0.82	12.20
105	20	0.71	19.29
120	20	0.62	32.26
135	10	0.5	20
150	20	0.38	52.63
165	10	0.3	33.33
180	10	0.25	40

For image zooming the computation time for different values of scaling factor as obtained for above image are as follows:

Table 6.2 Zooming test data for different scaling factors

Scaling Factor	Processing Time on CPU (ms)	Processing Time on 192 core GPU(ms)	Speed up obtained
1	0	0.29	0
2	40	1.69	23.67
3	120	4.27	28.10
4	220	7.87	27.54
5	340	13.5	25.19
6	510	18	28.33
7	700	23.6	29.67
8	910	30.79	29.56
9	1150	37.36	30.78
10	1420	48.94	29.02
11	1740	57.28	30.38
12	2080	67.25	30.93
13	2740	78.58	34.87
14	2840	90.56	31.36
15	3260	122.84	26.54

CHAPTER - 7

RESULTS AND DISCUSSION

The input and output of the image rotation operation is shown below:

Fig 7.1 Input Image for rotation

Fig 7.2 Image rotated by 90 degrees

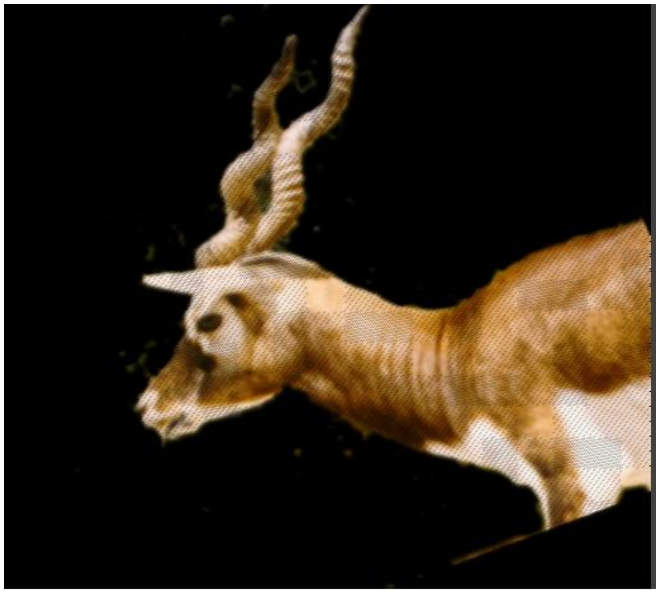


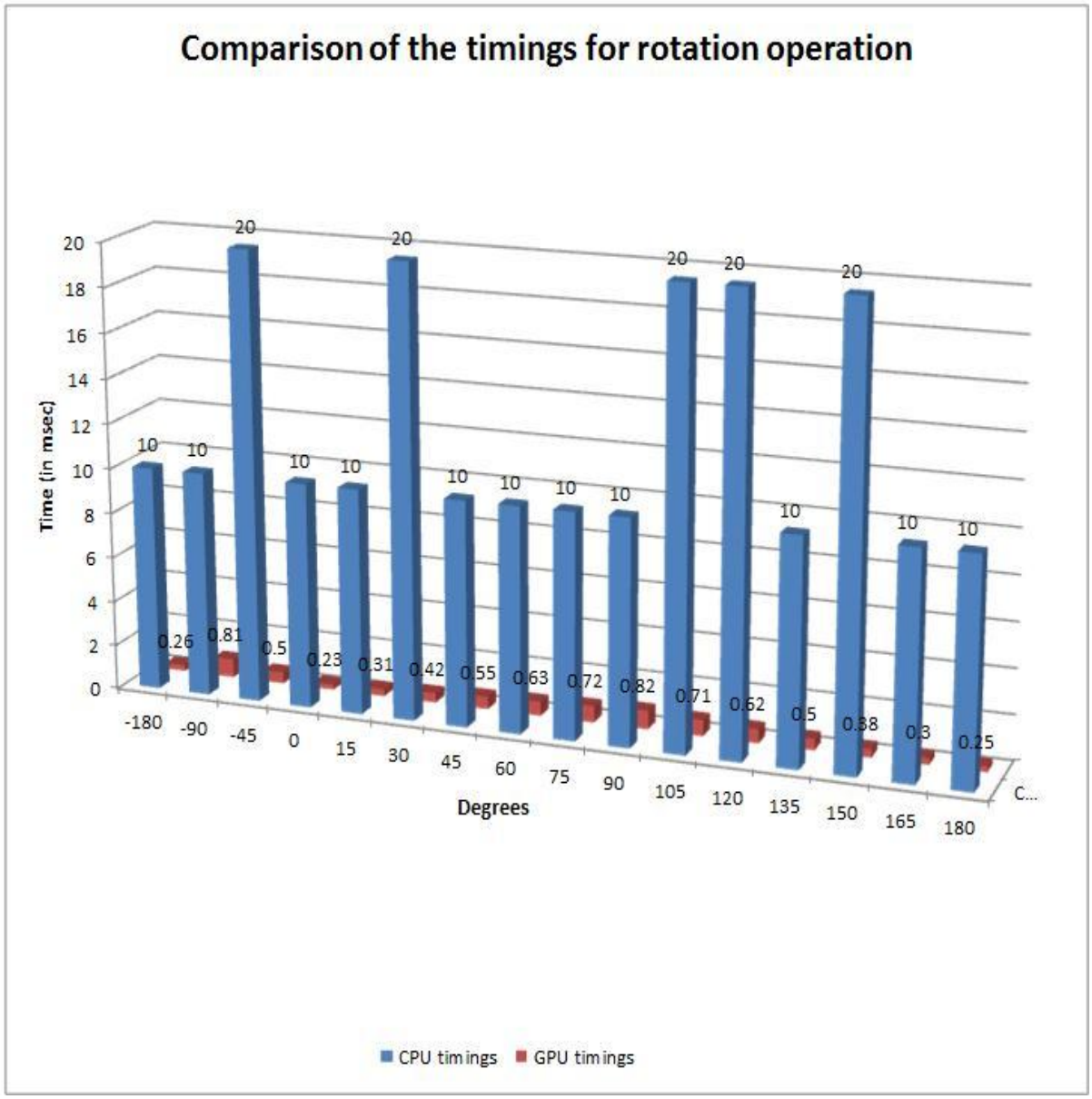
Fig 7.3 Image rotated by 45 degrees

The input and output images for image zooming are shown below:

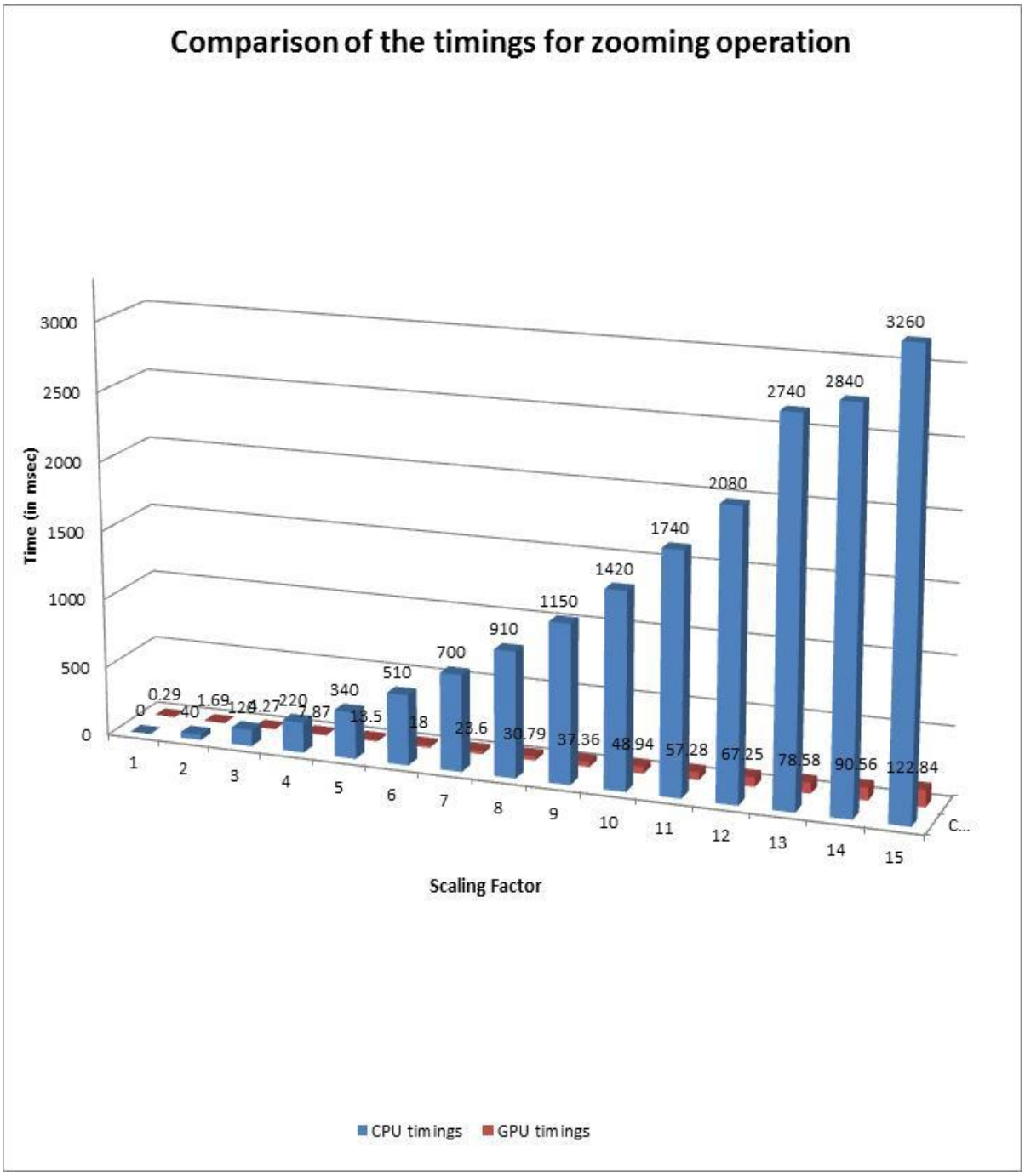
Fig 7.4 Input image (512X512 pixels) for zooming

Fig 7.5 Output Image (2 times scaled)

The results are analysed for different values of degree of rotation and scaling factor. Following graphs show the variation of the processing time for rotation by different angles of rotation and the variation of the processing time for zooming by different scaling factors respectively.



Graph 7.1 Comparison of processing time for rotation



Graph 7.2 Comparison for processing time for zooming

CHAPTER – 8
ADVANTAGES, APPLICATIONS AND LIMITATIONS

Merits:

It has been observed that the execution on GPU has brought about 30 times faster execution than on a CPU. Due to the parallel architecture and the use of parallel threads, all the operations have been done simultaneously on all the pixels. Hence the speedup obtained is about 30 times.

Demerits:

The architecture is not very efficient in case of small size images. This happens because of the extra time involved in memory transfer between the CPU and the GPU. This extra time reduces the overall efficiency of smaller size images.

Full utilization of the different memory units has not been obtained. There are many cached memory units that are yet to be used. Their usage will definitely increase the performance further.

Applications:

It can be embedded in software which deals in image rotation and zooming. The present implementation can provide highly efficient results in terms of execution timings and resources.

Many applications require image to be rotated or zoomed as their preliminary steps, this application can be used in those applications very efficiently. This will reduce the overall time of that application as well.

Limitations:

The image zooming algorithm is not working for floating point scaling factor.

The image produced after rotation has some amount of degradation which has been reduced to a greater extent, but still visible. The project is working only for PPM(P3) format images.

CHAPTER – 9

CONCLUSION

Parallel computing has been the new trend in the industry. This has been introduced to aid the existing serial computing. One of the most undesirable traits that serial computing has is the execution timings which are of greater magnitudes. This can make the overall working of the system slow. So parallel computing has helped in reducing the overall execution timings.

Image rotation and zooming are the basic methods that are very much used in various image processing applications. But the same problem that exists with serial execution is the magnitude of the timings. If there can be a mechanism through which this timing can be reduced then the overall application related with image processing will have better efficiency. This is where CUDA comes into scene.

CUDA provides an architecture that can implement the serial algorithms in parallel with greater efficiency. The methods such as image rotation and zooming are massively parallel processes. These operations can be executed in parallel achieving highly efficient results. This project demonstrates the difference between the execution timings of the CPU and the GPU. Exploiting the parallelism that exists in these operations, the processes have been exposed to parallel architecture. It has been demonstrated that the GPU produces the results roughly 30 times (on an average) faster than the CPU.

So, this parallel version of the methods can be implemented in the industry with many other applications. Parallel computing has revolutionized the industry and all such parallel applications have a bright future ahead.

CHAPTER – 10

FUTURE SCOPE

This project can further be improved for implementing certain extensions such as:

1. The zooming algorithm can be extended for floating point scaling factors. This is currently working for positive integers scaling factors.
2. Improve the quality of the image that is obtained after rotation of the image. There is still some degradation visible which can be removed.
3. Create a layer that can make other images of other formats work as well. Currently it is working only for PPM (P3) format images.
4. It can be extended for various other operations such as image segmentation, differentiation etc. As image rotation and zooming are the preliminary steps for many other applications, this can be extended.
5. Make the operation user friendly by creating an interface. This will help in useful interaction between the user and the application.

LIST OF FIGURES

LIST OF TABLES

LIST OF GRAPHS

REFERENCES

[1] Digital Image Processing by Gonzalez

[2] NVIDIA CUDA C Best Practices Guide

[3] NVIDIA CUDA C Programming Guide

[4] Real-Time Screen Image Scaling and Its GPU Acceleration by Wen Sun1, Yan Lu2, Feng Wu2, Shipeng Li2 (1, University of Science and Technology of China, Hefei, Anhui, 230027, China), (2, Microsoft Research Asia, Beijing, 100080, China)

[5] Midpoint Line Algorithm for High-speed High-accuracy Rotation of Images CHEN Xuede, LU Siwei, YUAN Xiaobu, Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1C 5S7

[6] Parallel Implementation For Image Rotation Using Parallel Virtual Machine (PVM) by J. Hinks, S.A. Amin, BIOCORE, Coventry University, Coventry, UK

[7] Hardware for Image Rotation by Twice Skew Transformations by Nu10 Tsuchida, Yoji Yamada, and Minoru Ueda

[8] Winscale: An Image-Scaling Algorithm Using an Area Pixel Model by Chun-Ho Kim, Si-Mun Seong, Jin-Aeon Lee, and Lee-Sup Kim

[9] http://homepages.inf.ed.ac.uk/rbf/HIPR2

[10] http://www.leptonica.com

[11] http://onlinelibrary.wiley.com/doi/10.1002/scj.4690200610/pdf

[12] www.howstuffswork.com

[13] www.wikipedia.com