MAULANA AZAD NATIONAL INSTITUTE OF TECHNOLOGY
Bhopal-462051

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MAJOR PROJECT REPORT

On

Image Histogram equalization using parallel processing (GPU Computation) in OpenCL to correct the contrast of images.

SUBMITTED IN PARTIAL FULFILMENT FOR THE DEGREE OF BACHELOR OF TECHNOLOGY

OF

MAULANA AZAD NATIONAL INSTITUTE OF TECHNOLOGY

By

Navratan Soni	091112023
Amit Kumar	091112107
Yugal Sharma	091112038
Bahar Uddin Mahmud	091112116

UNDER THE GUIDANCE OF

Dr. Nilay Khare

SESSION 2012-2013

DECLARATION

We hereby declare that the work which is been presented in this Major Project report entitled

Image Histogram equalization using parallel processing (GPU Computation) in OpenCL to correct the contrast of images

in partial fulfilment of requirements of the Final year major project in the field of Computer Science and Engineering is an authentic record of our own work carried out under the able guidance of **Dr. Nilay Khare**. The work has been carried out at **Maulana Azad National institute of Technology**.

The matter embodied in the project report has not been submitted for the award of any degree or deploma.

Name	Scholar no.	Signature
Navratan Soni	091112023	
Amit Kumar	091112107	
Yugal Sharma	091112038	
Bahar Uddin Mahmud	091112116	

MAULANA AZAD NATIONAL INSTITUTE OF TECHNOLOGY
Bhopal-462051

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that **Navratan Soni, Amit Kumar, Yugal Sharma and Bahar Uddin Mahmud,** students of Bachelor of Technology Final year (Computer science and Engineering), have successfully completed their project **"Image Histogram equalization using parallel processing (GPU Computation) in OpenCL to correct the contrast of images"** under my supervision in partial fulfilment of major project in Computer science and engineering.

Dr. Nilay Khare
(Project Guide)

Prof. Manish Pandey
(Project Coordinator)

SESSION 2012-2013

ACKNOWLEDGEMENT

With due respect we would like to express our sincere gratitude to our respected and learned guide **Dr. Nilay Khare** for his continuous motivation, valuable help and guidance. We are thankful to him for encouragement he has given to us in completing this project

Besides our Project guide, we would like to thank Prof. Manish Pandey (Project co-ordinator) for his encouragement, and insightful comments, during the project.

We are also thankful to all the other teachers and staff members of our department for their kind co-operation and suggesting improvements in the project.

We would like to express our deep appreciation towards our classmates for providing us the necessary suggestions in our project and cordial support

Navratan Soni

Amit Kumar

Yugal Sharma

Bahar Uddin Mahmud

Table of Contents

Abstract

Histogram equalization is a powerful way to correct the contrast of over exposed or under exposed images. These problems result to either high bright images or dark images. These problems can be solved by adjusting the contrast using histogram equalization. The histogram equalization is performed on intensity of images and it is done by performing the linear stretching of range of intensity of image to the range of 0-255, where 0 shows the black and 255 shows the white colour, and all the other 1-254 are the gray levels.

Mathematical formulation for this operation is given below:

$$\text{Output value} = 255 \times \frac{(\text{Input Value} - \text{Minimum Pixel value})}{(\text{Maximum Pixel value} - \text{Minimum Pixel value})}$$

Generally these operations are performed sequentially by reading image pixel-by-pixel. But if we apply the concept of parallel processing, the time consumed decreases dramatically. And the use of resource also increases. There are different framework available to complete the task in parallel manner. OpenCL is one of the frameworks for C/C++, Java and Python.

When we perform the task of processing pixels in parallel that makes use of GPU and capable CPU available on system, it gives a high increase in performance and resource uses.

1. Introduction

1.1 Overview

An image histogram is a type of histogram that acts as a graphical representation of the tonal distribution in a digital image. The horizontal axis of the graph represents the tonal variations, while the vertical axis represents the number of pixels in that particular tone. The left side of the horizontal axis represents the black and dark areas, the middle represents medium grey and the right hand side represents light and pure white areas. Thus, the histogram for a very dark image will have the majority of its data points on the left side and center of the graph. Conversely, the histogram for a very bright image with few dark areas and/or shadows will have most of its data points on the right side and center of the graph.

Image editors typically have provisions to create a histogram of the image being edited. Algorithms in the digital editor allow the user to visually adjust the brightness value of each pixel and to dynamically display the results as adjustments are made. Improvements in picture brightness and contrast can thus be obtained. **These image editors apply their image editing algorithms by continuously accessing image pixel by pixel and make a well equalized histogram.** Thus for very big and high resolution images, the time taken to manipulate pixels increases rapidly. The concept of parallel processing on GPUs and CPUs can decrease this time taken in manipulation of the images.

1.2 Advantage

This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values. The method is useful in images with backgrounds and foregrounds that are both bright or both dark.

Histogram equalization for high resolution images can cause high time and energy consumption thus equalization of histogram using parallel processing cause the saving of both as well as increases the resource uses.

1.3 Applications

- In the latest versions of different image editing software, this method is being used to manipulate the histogram of images.

- Latest version of Adobe Photoshop (CS6), uses parallel processing for every possible task to edit and create graphics and images.

- The method is useful in images with backgrounds and foregrounds that are both bright or both dark. In particular, the method can lead to better views of bone structure in x-ray images, and to better detail in photographs that are over or under-exposed.

- In scientific imaging where spatial correlation is more important than intensity of signal (such as separating DNA fragments of quantized length), the small signal to noise ratio usually hampers visual detection.

- This method automatically corrects the exposure of images, hence in automated systems for processing images.

2. Parallel Processing

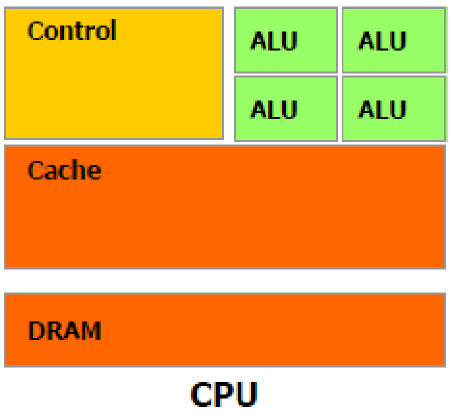
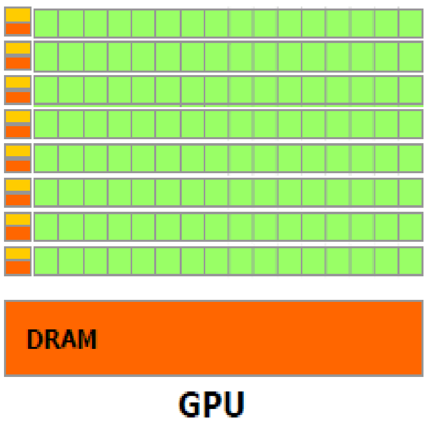
2.1 Introduction to parallel processing:

The simultaneous use of more than one processor to execute a program. Ideally, parallel processing makes a program run faster because there are more engines running it. In practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other. Most computers have just one CPU, but some models have several. There are even computers with thousands of CPUs.

In some applications size of input data is so large that even a low-order polynomial time algorithm surpasses the time limit. So one can use a number of processors to accomplish computational tasks concurrently and efficiently. But it's lengthy and costly. So Recently, GPU have found their places among general computing devices. General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research. GPUs are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations particularly linear algebra matrix operations.

GPU computing is the use of a GPU (graphics processing unit) together with a CPU to accelerate general-purpose scientific and engineering applications. CPU + GPU is a powerful combination because CPUs consist of a few cores optimized for serial processing, while GPUs consist of thousands of smaller, more efficient cores designed for parallel performance. Serial portions of the code run on the CPU while parallel portions run on the GPU.We are doing this processing using both CUDA and OpenCL. CUDA and OpenCL are both GPU programming frameworks that allow the use of GPUs for gen Therefore, in recent times, more focus has been put on the possibility of using Graphics Processing Units or GPU's for general-purpose computation, as the peak computational ability of these devices greatly exceeds those of even the best Central Processing Units or CPU's. Using GPU's to accelerate already existing algorithms is for that reason a very interesting prospect and it is thought that many applications with high performance demands in the future will make use of GPU's.

2.2 Comparison between Sequential processor and parallel processor

Sequential processor	parallel processor
Cores with low latency memory access.	High-performance many-core processors.
Optimized for sequential and branching algorithms.	Enormous parallel computing capacity algorithms
Ready for less workloads compared to GPU.	Ready for emerging workloads.
Less ALUs and no parallelism	More ALUs and massively parallel
Focuses on per thread performance	Throughput oriented
Runs existing applications very well.	Architecture well matched to media workloads: video, audio, graphics.
General CPU architecture:  *Figure 2.2(a): CPU architecture*.	**General GPU architecture:**  *Figure 2.2(b): GPU architecture*.

2.3 Parallel processing using OpenCL

OpenCL™ (Open Computing Language) is the first truly open and royalty-free programming standard for general-purpose computations on heterogeneous systems. OpenCL™ allows programmers to preserve their expensive source code investment and easily target multi-core CPUs, GPUs, and the new APUs. With OpenCL we can have many workers each executing a small piece of the work instead of a single worker doing all the job. The 1000 sums are executed at the same time, in parallel. After OpenCL came in action this parallel processing become more important and popular. For math and heavy calculation this technology would be a great combination of work.

2.4 Why OpenCL?

The OpenCL standard defines a set of data types, data structures, and functions that augment C and C++. Developers have created OpenCL ports for Java and Python, but the standard only requires that OpenCL frameworks provide libraries in C and C++.There is two chief advantages of OpenCL portability and parallel programming.

2.4.1 Portability

OpenCL adopts a similar philosophy of java "Write once, run on anything". Every vendor that provides OpenCL-compliant hardware also provides the tools that compile OpenCL code to run on the hardware. This means you can write your OpenCL routines once and compile them for any compliant device, whether it's a multicore processor or a graphics card. OpenCL applications can target multiple devices at once, and these devices don't have to have the same architecture or even the same vendor. As long as all the devices are OpenCL-compliant, the functions will run. This is impossible with regular C/C++ programming, in which an executable can only target one device at a time.

2.4.2 Parallel Programming:

OpenCL includes aspects of concurrency, but one of its great advantages is that it enables *parallel programming*. Parallel programming assigns computational tasks to *multiple* processing elements to be performed at the same time. In OpenCL parlance, these tasks are called *kernels.* A kernel is a specially coded function that's intended to be executed by one or more OpenCL-compliant devices. Kernels are sent to their intended device or devices by *host applications*. A host application is a regular C/C++ application running on the user's development system, which

We'll call the host. Kernels can also be executed by the same CPU on which the host application is running. Hosts applications manage their connected devices using a container called a *context*. To create a kernel, the host selects a function from a kernel container called a *program*. Then it associates the kernel with argument data and dispatches it to a structure called a *command queue*. The command queue is the mechanism through which the
host tells devices what to do, and when a kernel is queued, the device will execute
the corresponding function. An OpenCL application can configure different devices to perform different tasks, and each task can operate on different data. In other words, OpenCL provides full *task-parallelism*. This is an important advantage over many other parallel-programming toolsets, which only enable *data-parallelism*. In a data-parallel system, each device receives the same instructions but operates on different sets of data.

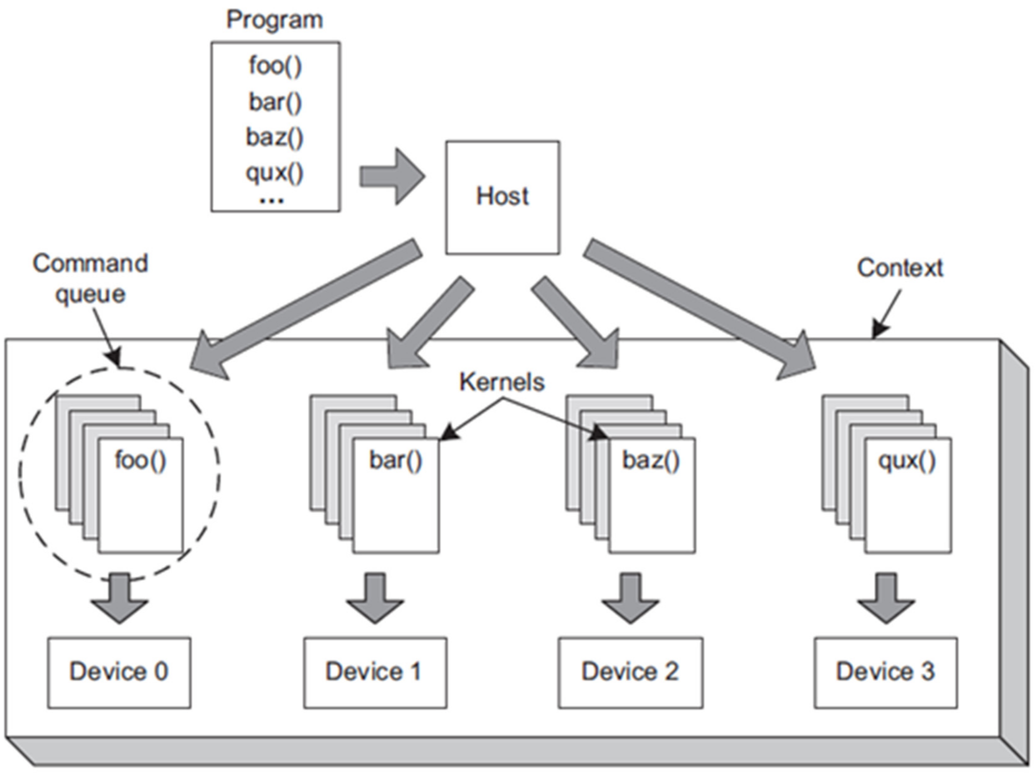


Figure 2.2: Kernel distribution among OpenCL compliant devices.

Portability, vector processing, and parallel programming make OpenCL more powerful than regular C and C++, but with this greater power comes greater complexity. In any practical OpenCL application, have to create a number of different data structures and coordinate their operation. It can be hard to keep everything straight.

2.5 GPU architecture:

In order to understand how to efficiently use a GPU it is necessary to have some knowledge on how it works and how its architecture looks. This chapter will thus serve to give an overview of the GPU architecture and describe how this is efficiently used in OpenCL.
If the architecture of a GPU is compared to that of a CPU, the main difference is that much more of the available chip transistors are devoted to processing instead of cache and flow control. Since CPU's needs to handle a large variety of complex problems and often at the same time, it must have a lot of registers and cache to keep track of the flow in an executing program. With the GPU on the other hand, the same program is executed for each data element, which means there is a much lower requirement for sophisticated flow control. Because the program is executed on many data elements and ideally has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

2.6 OpenCL programming model:

It is important to understand the OpenCL programming model in order to understand how to build OpenCL enabled applications which effectively utilize GPU's. The following figure below shows an illustration of the OpenCL programming model on a GPU.

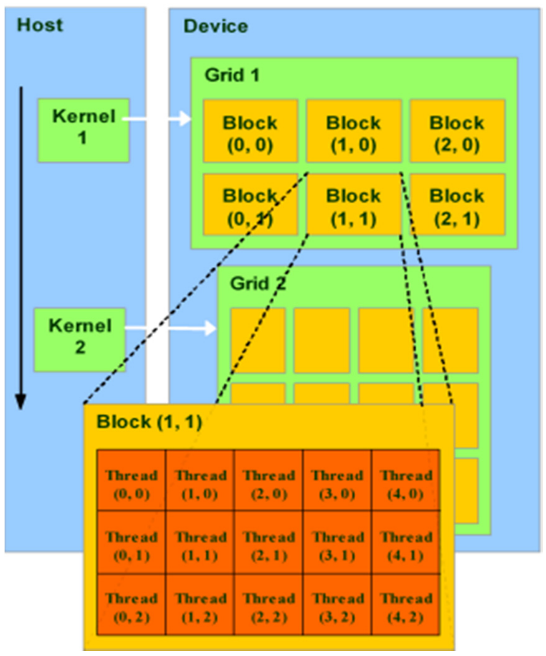


Figure 2.3: OpenCL programming model: showing CPU (host) to GPU (device) communication. Kernels are mapped to grids on the device, which execute blocks of threads concurrently.

When solving a problem using GPU's, one must be aware that the problem needs to be split into sub-problems which can be solved independently. The reason for this is, as explained earlier, that no memory exists for communicating between multiprocessors. This means that in order to utilize all multiprocessors on a GPU, the main problem must be divided into sub-problems, which can be solved on a single multiprocessor without communication to any other sub-problem.

The OpenCL programming model introduces a grid containing a number of blocks, which in turn contains a number of threads as illustrated in figure. A block is always executed on one specific multiprocessor, which means that threads within a block can communicate with each other, whereas threads within different blocks cannot communicate with each other. A block has I unique identifier in 2D, defining its position on the grid. Likewise, the threads within blocks have unique identifiers in 3D, which defines their position in the block. To give an example of how this is suitable for solving a problem with many independent calculations, a picture with 64x64 pixels is considered. It would then be possible to define 4 blocks in the x direction and 4 blocks in the y direction each containing 16 threads in the x-

and y- direction. This way a single thread would be defined for each pixel in the image. A kernel is then defined, which is a program stub, which will be executed by each defined thread. Based on the block- and thread identifier each thread knows which pixel in the image it is supposed to work on. As up to 480 simple processors reside on the GPU used in this project, the image of 64x64 could be manipulated in only very few clock cycles.

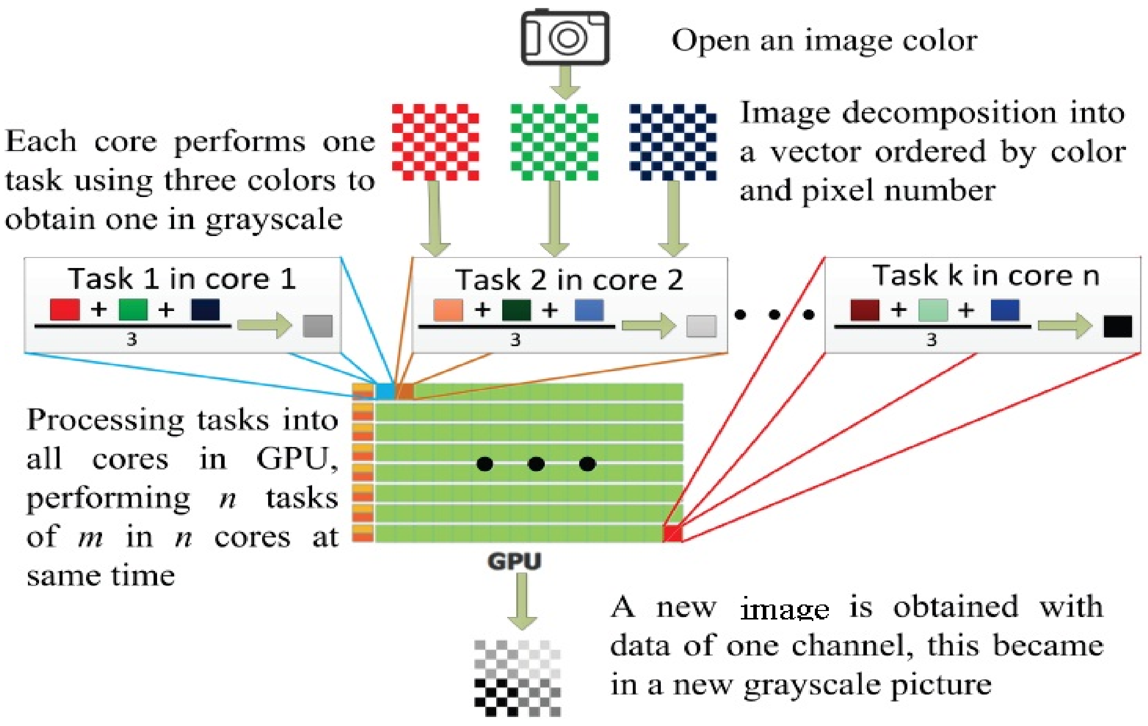
2.7 Using parallelism in image applications

When solving a problem using GPU's, one must be aware that the problem needs to be split into sub-problems which can be solved independently. The reason for this is that no memory exists for communicating between multiprocessors. This means that in order to utilize all multiprocessors on a GPU, the main problem must be divided into sub-problems, which can be solved on a single multiprocessor without communication to any other sub-problem.

Many image processing algorithms are computationally expensive and parallelizable. Moreover, traditional processing methods can not satisfy real time requirement for large size image processing. GPU is only useful for extremely data parallel workloads, where similar calculations are executed on quantities of data that are arrayed in a regular grid-like fashion, so it is one of ideal solutions of large size images.

Computer Vision algorithms are particular suitable to be executed on GPU's because of their general nature. In most cases many equal computations are done for each pixel in an image. These computations are often independent meaning they can easily be computed in parallel. Where a CPU will compute an output for each pixel in an image sequentially, a GPU is able to compute the output of hundreds of pixels at the same time. This result in much faster overall computation of an image output, thus making the use of GPU's very feasible in many Computer Vision algorithms.

How Parallelism improves the performance of image enlargement is described in the picture next page, this picture is illustrating how GPU with n core enhance the speed n-times over sequential processing of image.



Open an image color
Image decomposition into a vector ordered by color and pixel number
Each core performs one task using three colors to obtain one in grayscale
Task 1 in core 1
Task 2 in core 2
Task k in core n
3
Processing tasks into all cores in GPU, performing n tasks of m in n cores at same time
GPU
A new image is obtained with data of one channel, this became in a new grayscale picture

Figure 2.4: Figure showing how parallelism is used on GPU when images are used.

2.8 Advantages of parallel programming

Parallel processing is much faster than sequential processing when it comes to doing repetitive calculations on vast amounts of data. This is because a parallel processor is capable of multithreading on a large scale, and can therefore simultaneously process several streams of data. This makes parallel processors suitable for graphics cards since the calculations required for generating the millions of pixels per second are all repetitive. GPUs can have over 200 cores to help them in this. The CPU of a normal computer is a sequential processor - it's good in processing data one step at a time. This is needed in cases where the calculation the processor is performing depends on the result of the previous calculation and so on; in parallel processing these kinds of calculations will slow it down, which is why CPUs are generally optimized for sequential operations and have only 1-8 cores.

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously

Use of non-local resources: Using compute resources on a wide area network, or even the Internet when local compute resources are scarce.

Limits to serial computing: Both physical and practical reasons pose significant constraints to simply building ever faster serial computers

3. Histogram Equalization and Algorithm

3.1 Histogram equalization

There are situations where we would want to reveal detail in an image that cannot easily be seen with the naked eye. One of the several techniques to enhance an image in such a manner is histogram equalization, which is commonly used to compare images made in entirely different circumstances. Extreme examples may include comparisons of photographs with different exposures, lighting angles and shadow casts.

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. A good histogram is that which covers all the possible values in the gray scale used. This type of histogram suggests that image has good contrast and the details in the image can be observed more easily. In histogram equalization we are trying to maximize the image contrast by applying a gray level transform which tries to flatten the resulting histogram. It turns out that the gray level transform that we are seeking is simply a scaled version of the original image's cumulative histogram.

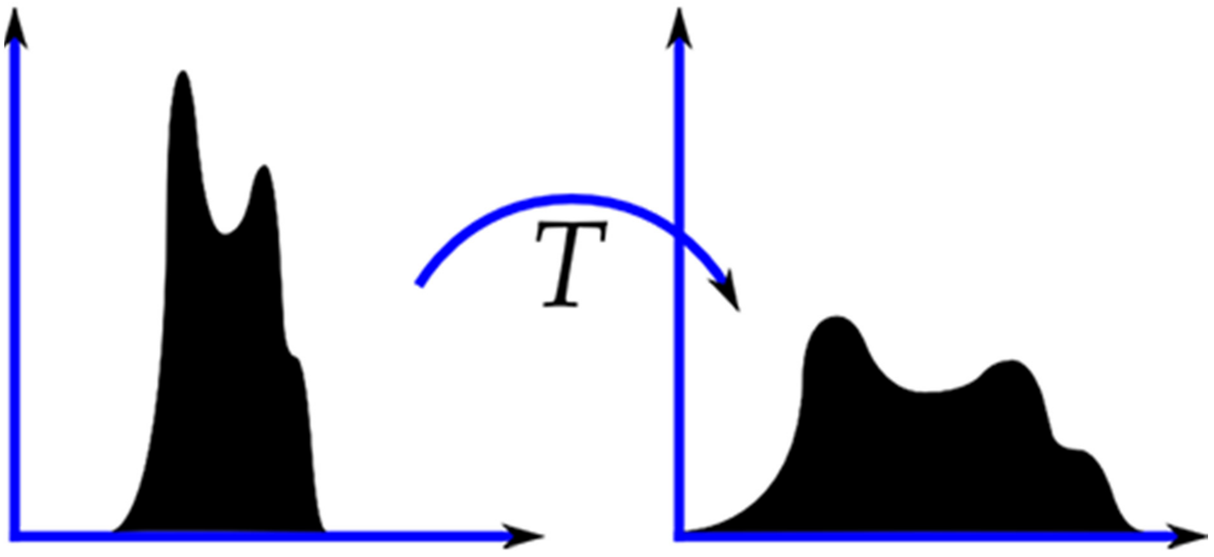


Figure 3.1: Histogram Equalization

3.2 Histogram equalization algorithm

The concept of histogram equalization is to spread otherwise cluttered frequencies more evenly over the length of the histogram. Frequencies that lie close together will dramatically be stretched out. These respective areas of the image that first had little fluctuation will appear grainy and rigid, thereby revealing otherwise unseen details.

A histogram equalization algorithm will determine the ideal number of times each frequency should appear in the image and, theoretically, re-plot the histogram appropriately. However, because image data isn't stored in an analogue manner, this is usually not possible. Instead, the image data is stored digitally and limited to n bits of color depth, thus the image cannot be requantified to meet our requirement.

Nevertheless, by ensuring that the number of times a frequency in a certain range remains as close to the ideally equalized histogram as possible, we can work around the issue of re-quantification. The solution is simple and elegant.

The ideal number of pixels per frequency *i* is the total number of pixels in the image divided by the total number of possible image frequencies N. The algorithm counts the frequencies from 0 to N and shifts as many pixel frequencies into that position as long as this number of pixels is less than or equal to a certain delimiter that increases linearly to the frequency. If a pixel frequency doesn't fit, it is pushed to the right along the horizontal axis until a place is found.

Histogram equalization can be done using two methods, Linear stretching and using cumulative distribution function. **We are using Linear stretching method to equalize the histogram.**

Linear Stretching
In this method we use the stretch operation which re-distributes values of an input map over a wider or narrower range of values in an output map.

1. The input values of a map are re-scaled to output values in the output map.
2. Input values are specified by the 'stretch from' values; the lower and upper 'stretch from boundary values are included in the stretching.

3. Output values are specified by the output domain and the value range and precision of this domain.

We need to stretch the input histogram to desired output histogram value (0-255).

The input values of a map are re-scaled to output values in the output map. Input values are specified by the 'stretch from' values; the lower and upper 'stretch from' boundary values are included in the stretching. Output values are specified by the output domain and the value range and precision of this domain.

So to map the levels, we use

$$\frac{(\text{inVal}-\text{inLow})}{(\text{inHigh}-\text{inLow})} = \frac{(\text{opVal}-\text{opLow})}{(\text{ophigh}-\text{opLow})}$$

$$\text{opVal} = \text{opLow} + \frac{(\text{inVal}-\text{inLow}) * (\text{ophigh}-\text{opLow})}{(\text{inHigh}-\text{inLow})}$$

Where,

opVal = Value of pixel in output map   inVal = Value of pixel in input map

inLow = Lower value of 'stretch from' range   inHigh = Upper value of 'stretch from'

opLow = Lower value of 'stretch to' range = 0 here

opHigh = Upper value of 'stretch to' range = 255 here

When the 'stretch from' range is specified as values, these are inLow and inHigh.

All input values smaller than or equal to inLow are brought to opLow, and all input values greater than or equal to inHigh are brought to opHigh. When choosing output domain `Image`, then opLow is 0, and opHigh is 255. When choosing domain Value, the minimum and maximum values specified as the value range are used as opLow and opHigh ('stretch to' range).The value specified as the precision of the output value domain is used to round opVal.

The result of a linear stretch is that all input values are stretched to the same extent, see Fig. 1 below.

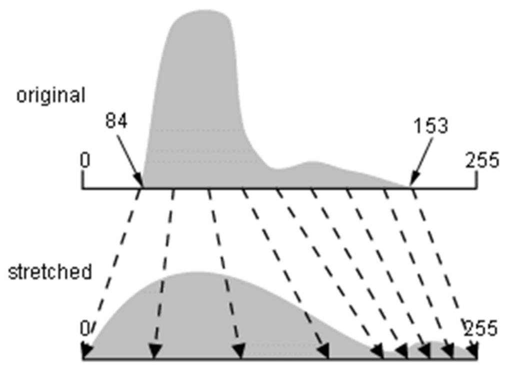


Figure 3.2: Histogram stretching

So the final usable formula can be derived as

$$\text{opVAl} = \frac{(\text{inVal} - \text{inLow})}{(\text{inHigh} - \text{inLow})} \times 255$$

Then the output value is rounded-off to get the new value in the matrix.

Consider one matrix

52	60	74	75	86
88	92	95	100	102
108	109	113	118	128
132	140	142	148	154
170	182	191	200	206

inLow here = 52, and inHigh here is 206 so the final calculate matrix shown will be stretched from 0 to 255, hence resultant matrix values will be calculated as.

e.g. for input value 113 op value will be = 255*(113-52)/(206-52) = 101

So the final calculated matrix will be as

0	13	36	38	56
60	66	71	79	83
93	94	101	109	126
132	146	149	148	159
195	182	215	245	255

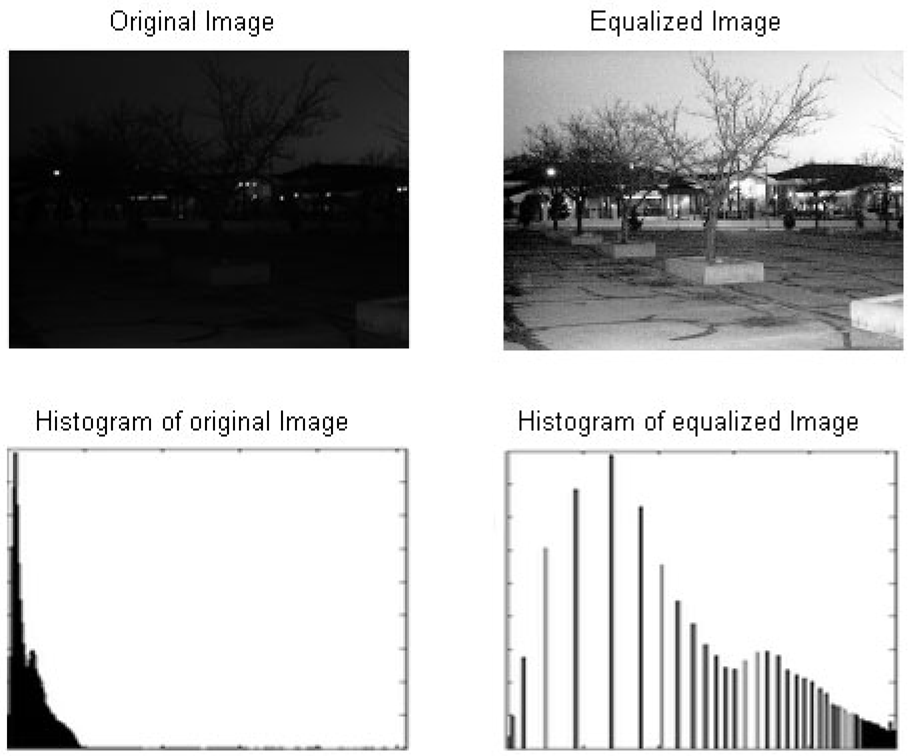
This data is calculated in parallel when we pass this data on to the kernel function.

3.3 Uses of histogram equalization

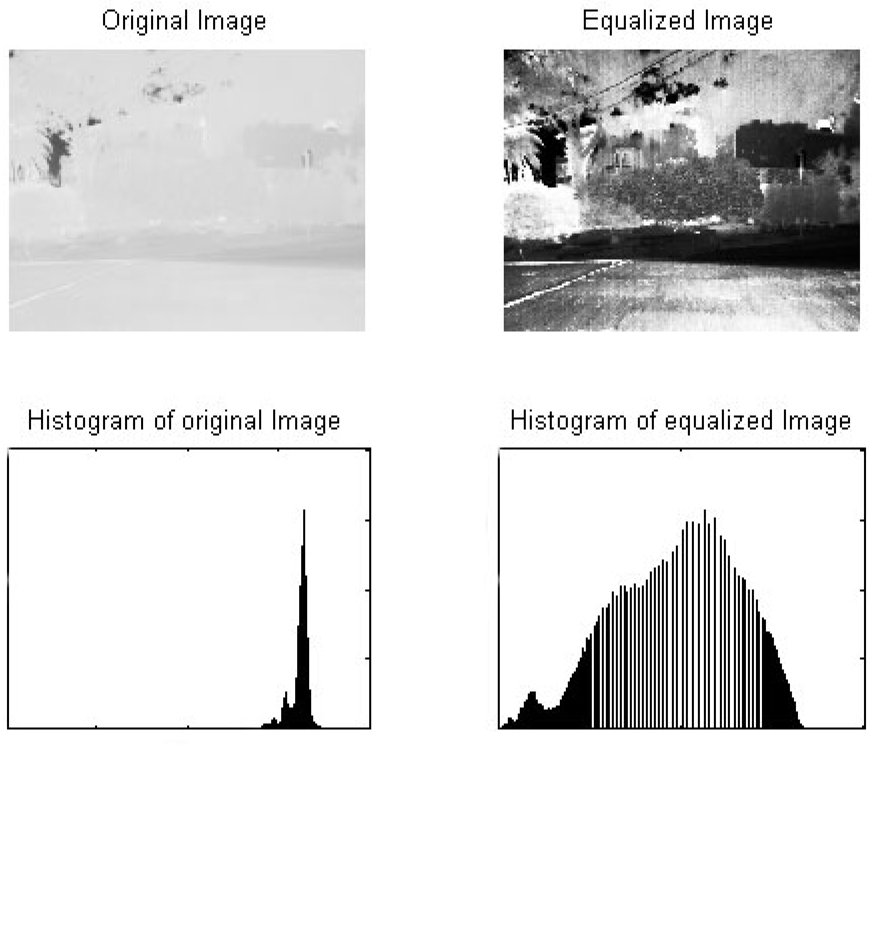
Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values. So it brightens under exposed images and darkens over exposed images.

The above describes histogram equalization on a grayscale image. However it can also be used on color images by applying the same method separately to the Red, Green and Blue components of the RGB color values of the image. However, applying the same method on the Red, Green, and Blue components of an RGB image may yield dramatic changes in the image's color balance since the relative distributions of the color channels change as a result of applying the algorithm.

Histogram equalization of images with under exposure



Histogram equalization of images with over exposure



4. Parallel implementation and results

4.1 Introduction

The implementation is done in three steps

1. Generation of intensity matrix of input image using MATLAB.
2. Data is read from pixels and passed onto the processing unit where kernel functions perform the parallel processing on data and perform the resultant result matrix is generated.
3. Generation of result image using the result matrix in MATLAB.

4.2 Parallel implementation of algorithm

To process the data in parallel manner, we convert the data into 1 dimensional stream and pass it onto the parallel processing unit. The passed data then, according to the capacity of parallel processor, is processed batch-by-batch. The size of the batch is same as the GPUs capacity of processing data at one time parallel, so the batches of data are processed sequentially and the data in every batch is then processed parallel manner.

For one batch of data, the every work item gets its own personal id in kernel function using *"get_global_id(0)"* and then every work item is processed on one of the cores of GPU.

Consider one example of addition of two vectors, one batch from each of two, each data is added in parallel manner.

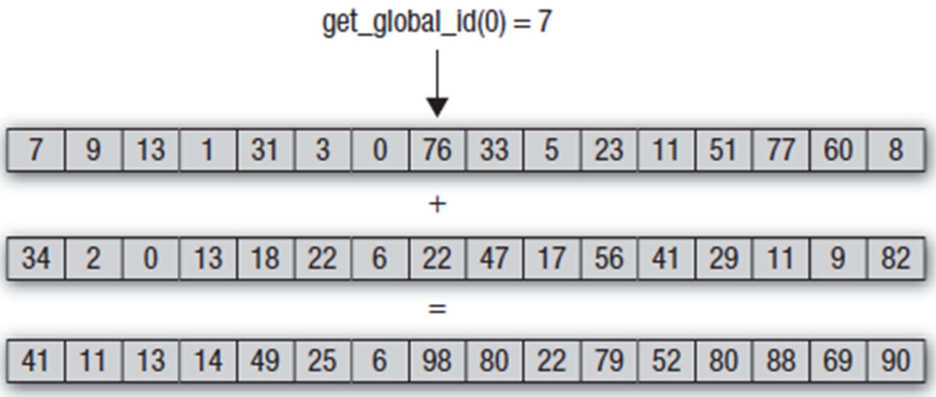


Figure 4.1 example of parallel addition of two vectors

Finding the possibilities in algorithm, where the parallelism can be done:

In order find possibilities in the algorithm, where the parallelism can be applied, we find three steps where one part of the data is independent of other data, as in here we are calculating the new value for every pixel, the **output value of one pixel is dependent** on the minimum value pixel, and maximum value pixel. If this is modified at any step than we can spoil our calculation, that's why we store both of the values in two different constant integers to calculate the output value of every new pixel.

We can apply parallel processing here in

- Reading the data from matrix, as every pixel is independent
- Processing the data and calculating the new values
- Again writing data simultaneously in output buffer

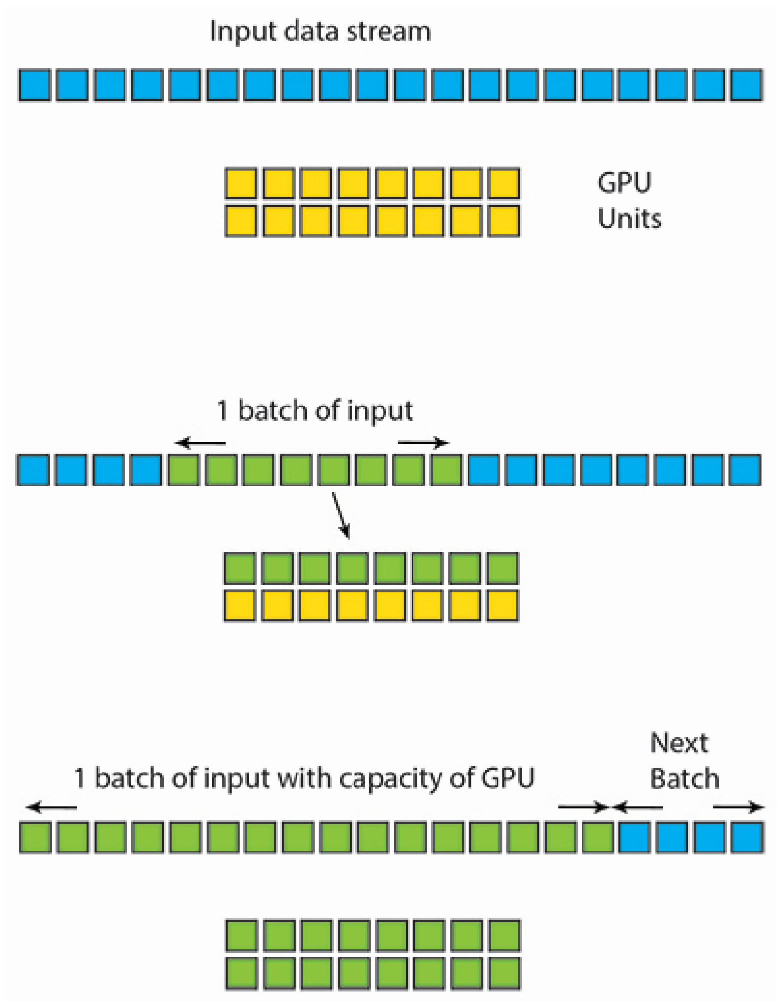


Figure 4.2 Passing the data on GPU

4.3 Programming implementation in OpenCL

A. C++ Code **-** The following code shows the program build over C++ using OpenCL frame work and this program passes the input to the kernel function and get the output of kernel file

```
#include <stdlib.h>
#include <sys/types.h>
#include <CL/cl.h>
#include <iostream>
#include <time.h>

#define WIDTH 10  // defining the width of demo matrix
#define HEIGHT 10  // defining the height of demo matrix
int main()
{
#ifdef CL_DEVICE_TYPE_CPU
       std::cout << "This program is running parallely on CPU" << "\n";
#endif
       cl_platform_id platform;
       cl_context context;
       cl_command_queue queue;
       cl_device_id device;
       cl_int error;

       if (clGetPlatformIDs(1,&platform, NULL) != CL_SUCCESS) {
              std::cout << "Error getting platform id\n";
              exit(error);
              }
       std::cout << "PLatform Id :" << platform <<"\n";

       // Device
       // to run the program on CPU write CL_DEVICE_TYPE_CPU instead of
CL_DEVICE_TYPE_GPU
       if (clGetDeviceIDs(platform, CL_DEVICE_TYPE_CPU, 1, &device, NULL)!=
CL_SUCCESS) {
              std::cout << "Error getting device ids\n";
              exit(error);
              }
       std::cout << "Device Id :" << device <<"\n";

       // Context
       context = clCreateContext(0, 1, &device, NULL, NULL, &error);
       if (error != CL_SUCCESS) {
           std::cout << "Error creating context\n";
           exit(error);
              }

       //queue
       queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE,
&error);
       if (error != CL_SUCCESS) {
              std::cout << "Error creating command queue\n";
              exit(error);
              }
```

```
	// Initializing the input matrix for demo
	// for large inputs, the matrix should be initialized by reading input file
	int ipMat[WIDTH][HEIGHT] = 		{{40,60,80,100,120,140,160,180,200,220},
						{60,90,120,150,180,210,240,14,44,74},
						{80,120,160,200,240,24,64,104,144,184},
						{100,150,200,250,44,94,144,194,244,38},
						{120,180,240,44,104,164,224,28,88,148},
						{140,210,24,94,164,234,48,118,188,2},
						{160,240,64,144,224,48,128,208,32,112},
						{180,14,104,194,28,118,208,42,132,222},
						{200,44,144,244,88,188,32,132,232,76},
						{220,74,184,38,148,2,112,222,76,186}};

	// computing total size of image's matrix
	const int totalImgSize = (HEIGHT)*(WIDTH);
	std::cout << "This is a break point to calculate image size :" << totalImgSize 
<< "\n";

	// Computing the size of buffer needed to store the image(matrix) data
	const int buffSize = sizeof(int)*totalImgSize;

	//dynamically creating 1D matrix (Arrys) to store input and output 2D matrix
	int* ipArray = new int[totalImgSize];
	int* opArray = new int[totalImgSize];

	//Storing input matrix to input 1D array and finding Maximum and Minimum
	int p = 0;
	int min=ipMat[0][0],
		max=ipMat[WIDTH-1][HEIGHT-1];

	for(int i=0; i<HEIGHT; i++)
	{
		for(int j=0; j<WIDTH; j++)
		{
			ipArray[p]=ipMat[i][j];
			if (min > ipArray[p])
				min=ipArray[p];
			if (max < ipArray[p])
				max=ipArray[p];
			p++;
		}
	}

	std::cout << "minimum value is :" << min <<"\n";
	std::cout << "mxaimum value is :" << max <<"\n";

	//Creating pointers to values (Only pointers can be passed while setting kernel 
arguments)
	int *minp = &min;
	int *maxp = &max;

	//creating three buffers, two for the same image and one for the output
	cl_mem ipBuffer = clCreateBuffer(context, CL_MEM_READ_ONLY | 
CL_MEM_COPY_HOST_PTR, buffSize, ipArray, &error);
	cl_mem opBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, buffSize, NULL, 
&error);

	//Creating program from reading the .cl file
	cl_program program;
	FILE *program_handle;
	char* program_buffer;
	size_t program_size;
```

```
        program_handle = fopen("processMatrix.cl", "r");
        fseek(program_handle, 0, SEEK_END);
        program_size = ftell(program_handle);
        rewind(program_handle);
        //std::cout<<program_size<<"\n";
        program_buffer = (char*)malloc(program_size + 1);
        program_buffer[program_size] = '\0';
        fread(program_buffer, sizeof(char),program_size,program_handle);
        fclose(program_handle);

        program = clCreateProgramWithSource(context, 1,(const char**)&program_buffer,
&program_size, &error);

        // Builds the program
        error = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

        // Extracting the kernel
        cl_kernel matProc = clCreateKernel(program, "procMat", &error);

        //Passing kernel arguments to run the kernel function
        error = clSetKernelArg(matProc, 0, sizeof(cl_mem), &ipBuffer);
        error |= clSetKernelArg(matProc, 1, sizeof(cl_mem), &opBuffer);
        error |= clSetKernelArg(matProc, 2, sizeof(size_t), minp);
        error |= clSetKernelArg(matProc, 3, sizeof(size_t), maxp);

        //Launching kernel and defining size of batch that is processed at one time
        const size_t work_units_per_kernel = (size_t)totalImgSize;
        clEnqueueNDRangeKernel(queue, matProc, 1, NULL,  &work_units_per_kernel, NULL,
0, NULL, NULL);

        // For time Profiling
        cl_event timing_event;
        cl_ulong time_start, time_end;
        float read_time;

        //reading the output and putting from opBuffer buffer to opArray outputArray
        clEnqueueReadBuffer(queue, opBuffer, CL_TRUE, 0, buffSize, opArray, 0, NULL,
&timing_event);
        clGetEventProfilingInfo(timing_event, CL_PROFILING_COMMAND_START,
        sizeof(time_start), &time_start, NULL);
        clGetEventProfilingInfo(timing_event, CL_PROFILING_COMMAND_END,
        sizeof(time_end), &time_end, NULL);
        read_time = time_end - time_start;
        printf(".............%f(miliseconds)................\n\n",(read_time/1000000)
);

        //Giving the output of 1-D stram output
        for(int i = 0; i < totalImgSize; i++)
               std::cout << opArray[i]<<"\t";

        // Cleaning up
        delete[] ipArray;
        delete[] opArray;
        clReleaseKernel(matProc);
        clReleaseCommandQueue(queue);
        clReleaseContext(context);
        clReleaseMemObject(ipBuffer);
        clReleaseMemObject(opBuffer);
        return 0;
}
```

B. kernel code – **(processMat.cl)** This code runs on the parallel processor and takes input in batches and provides the output by processing the data parallel.

```
__kernel void procMat (__global const int* src_a,__global int* res,const int min,const
int max){
        // getting the id for every work item
        const int idx=get_global_id(0);
        res[idx]=(255*(src_a[idx]-min))/(max-min);
      }
```

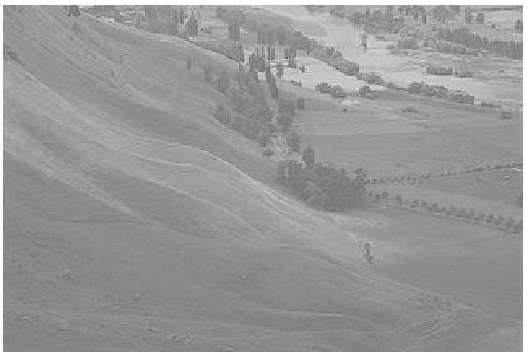
4.4 Results and Statistics

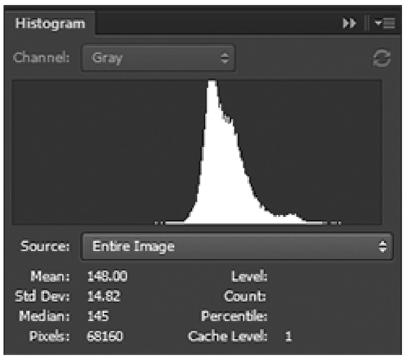
One of the image with resolution 320x213 is passed in MATLAB to generate the intensity matrix, and then matrix is sent to process parallel and then processed output matrix is passed to MATLAB again to generate the output image.

The histogram of input and output image is generated using the photo editing software.

4.4.1 Statistics of input image matrix

Width = 320 pixels	Lowest intensity value = 116
Height = 213 pixels	Highest intensity value = 205


Input Image



Channel: Gray

Source: Entire Image

Mean: 148.00
Std Dev: 14.82
Median: 145
Pixels: 68160
Level:
Count:
Percentile:
Cache Level: 1

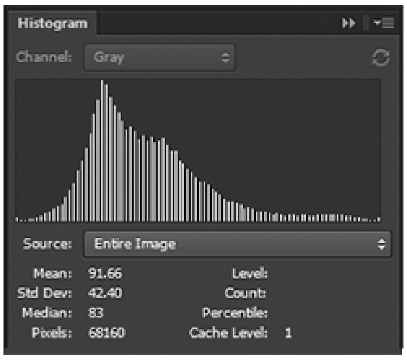
Input Histogram

4.4.2 Statistics of output image matrix

Width = 320 pixels   Lowest intensity value = 0

Height = 213 pixels   Highest intensity value = 255

Outpu Image

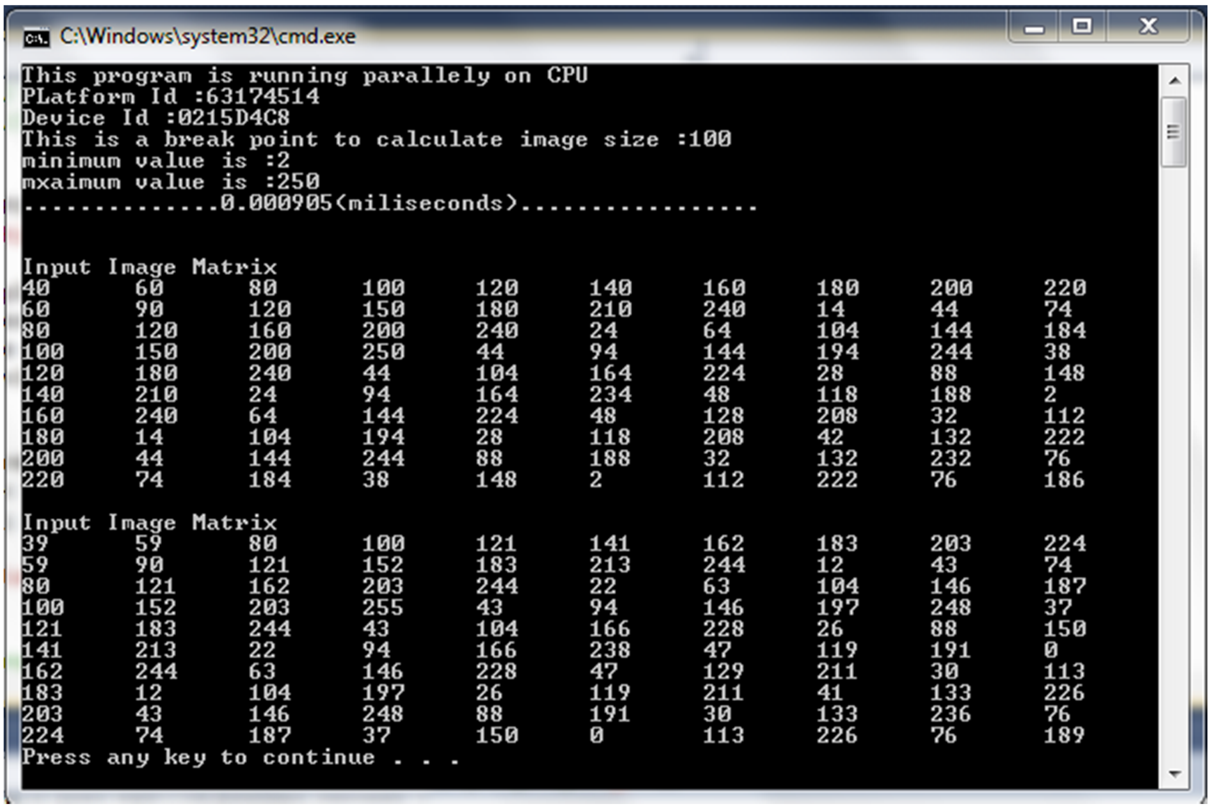


Source: Entire Image

Mean: 91.66 Level:
Std Dev: 42.40 Count:
Median: 83 Percentile:
Pixels: 68160 Cache Level: 1

Output Histogram

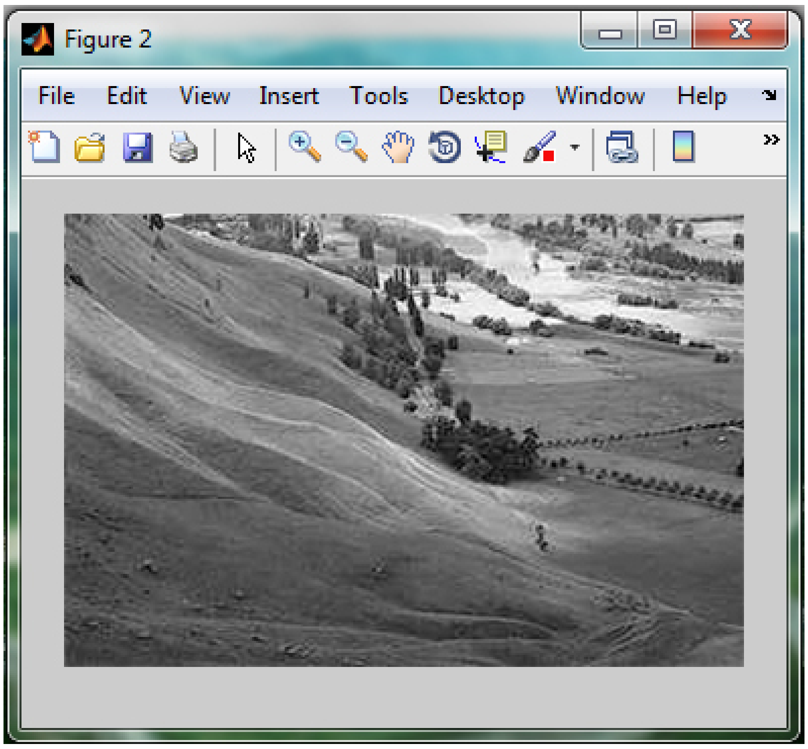
4.4.3 Snippets of project outputs



```
This program is running parallely on CPU
PLatform Id :63174514
Device Id :0215D4C8
This is a break point to calculate image size :100
minimum value is :2
mxaimum value is :250
..............0.000905(miliseconds)................


Input Image Matrix
40      60      80      100     120     140     160     180     200     220
60      90      120     150     180     210     240     14      44      74
80      120     160     200     240     24      64      104     144     184
100     150     200     250     44      94      144     194     244     38
120     180     240     44      104     164     224     28      88      148
140     210     24      94      164     234     48      118     188     2
160     240     64      144     224     48      128     208     32      112
180     14      104     194     28      118     208     42      132     222
200     44      144     244     88      188     32      132     232     76
220     74      184     38      148     2       112     222     76      186

Input Image Matrix
39      59      80      100     121     141     162     183     203     224
59      90      121     152     183     213     244     12      43      74
80      121     162     203     244     22      63      104     146     187
100     152     203     255     43      94      146     197     248     37
121     183     244     43      104     166     228     26      88      150
141     213     22      94      166     238     47      119     191     0
162     244     63      146     228     47      129     211     30      113
183     12      104     197     26      119     211     41      133     226
203     43      146     248     88      191     30      133     236     76
224     74      187     37      150     0       113     226     76      189
Press any key to continue . . .
```



4.4.3 Timing Comparison with different processors and data

We are using profiling in application so it allows us to evaluate the performance of computing hardware and coding methods. We get an estimate of processing the the data on CPUs and GPUs.

The outputs for different data inputs, can be calculated

S. No.	**Size of Input Matrix**	**Execution Timing on CPU (Parallel processing)(ms)**	**Execution Timing on GPU (Parallel Processing)(ms)**	**Execution time on CPU (sequentially Processing)(ms)**
1.	100*100	0.005	0.069	0.85
2.	200*200	0.026	0.076	0.92
3.	300*300	0.072	0.178	1.98
4.	400*400	0.159	0.279	3.23
5.	500*500	0.248	0.351	4.81

One thing can be noted from this table that parallel processing **on CPU is taking less time to execute the instructions, than GPU**. This is because when we have small amount of data, than time taken on CPU parallel processing is lower as we are using the global memory. But when very high amount of data is passed than time taken by GPU is less. We can see from the table as well that rate of growth of time consumption is higher in CPU.

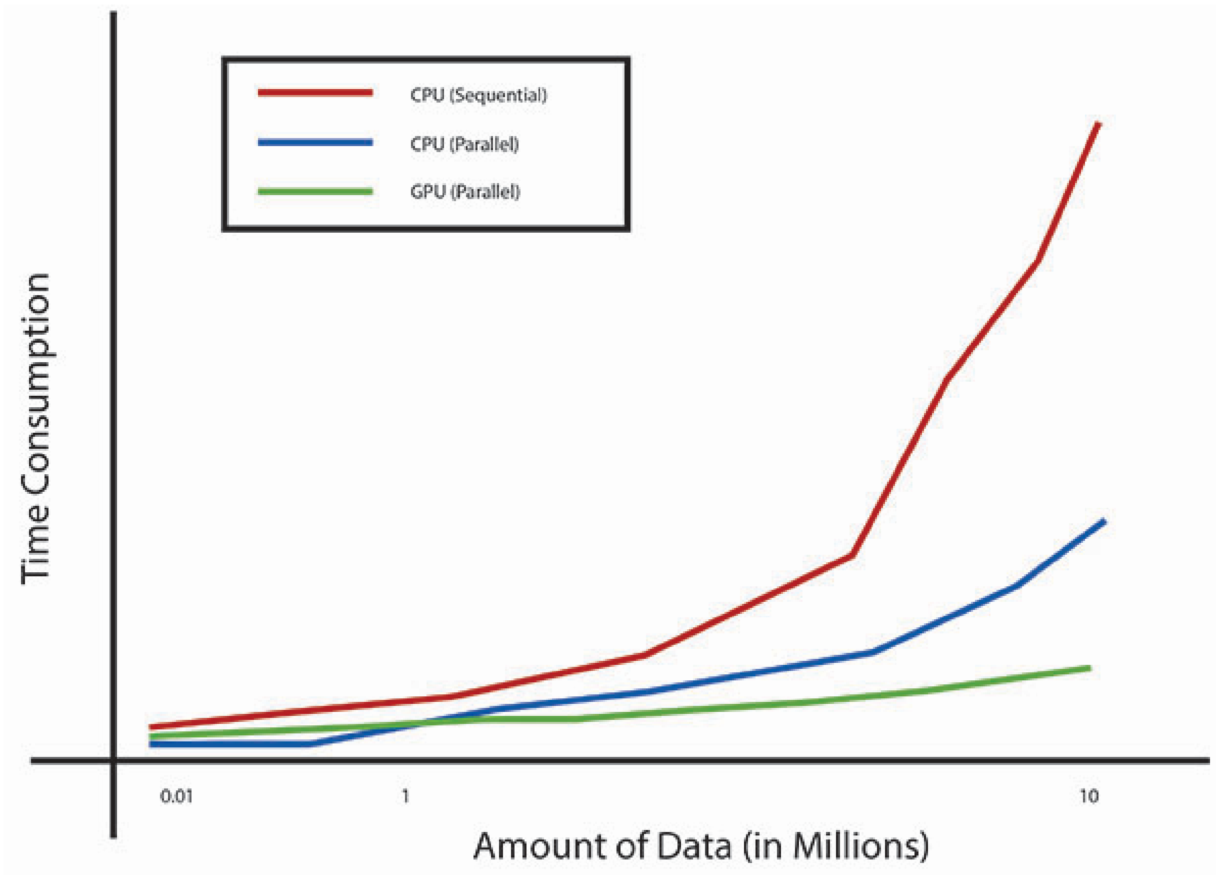


Fig 4.3 Comparison of time consumption v/s amount of data of different processors

4.5 Conclusion

Parallel processing of image histogram dramatically reduce the time consumption over the high resolution images, when ran on GPUs in comparison to CPUs. The results shows that the time consumption by CPUs on increasing the input data causes the timing increase in almost an exponential manner. But when processed on the GPUs and also on capable new CPUs in parallel the time increased is almost linear and very low on the high data input in comparison to the sequential processing.

These type of algorithm processing in parallel manner on data can be extended in many other areas and the processing can be sandwiched to increase the response rate of applications.

In this project the histogram equalization was implemented in parallel processing manner using OpenCL and results were compared on different GPUs and CPUs.

4.6 Future work and other extensions

- This system can be included in different image editing software available in market that can make use of GPUs.

- As this algorithm automatically corrects the grayscale exposure. This can be used to make some automated systems for parallel processing multiple images, and by using the some part of GPU for every image, these one image's data can also be processed in parallel.

- This system can be used to increase the contrast in x-ray to adjust the exposure of bone structure. Also other medical reports that include images can be corrected.

- The system can be used in correcting images taken by spatial satellites, as due to different environmental situations, image exposure may not be good, and taking millions of high resolution picture again can cause time as well as economical loss. Applying this method to process parallel can save time and money as well.

- Extending algorithms based on same concepts can also be implemented to make powerful tools to manipulate the colour images as well and can be extended to different image formats like RGB, Lab Colour Mode, HSL and CMYK.

5. References

- OpenCL-based design methodology for application-specific processors by De La Lama, C.S.; Huerta, P.; Takala J.H., Jaaskelainen P.O., Embedded Computer Systems (SAMOS), 2010 International Conference on Dated 19-22 July,2010, page.223-230.
- Image parallel processing based on GPU by Nan Zhang, 2nd International Conference on Advanced Computer Control (ICACC), 2010, page.367-370.
- Bräunl, T., with Feyrer, S., Rapf, W., Reinhardt, M.,Parallel Image Processing, Springer Verlag,Heidelberg, 2000.
- OpenCL in Action by Matthew Scarpino ISBN 978-1-61729-017-6.
- OpenCL Programming Guide by Aftab Munshi, Benedict R. Gester, Timothy G. Mattson, James Fung and Dan Ginsberg ISBN 978-0-321-74964-2.
- Stretch algorithm analysin in image histogram. Website :http://spatial-analyst.net/ILWIS/htm/ilwisapp/stretch_algorithm.htm.
- Khronos group, http://www.khronos.org/opencl
- Nvidia developer Zone, https://www.developer.nvidia.com
- AMD Developer Center, http://www.developer.amd.com
- Wikipedia, http://www.wikipedia.org