Image Histogram Equalization using Parallel Processing (GPU computation) in Open CL to correct the Contrast of Images

Bahar Uddin Mahmud*, Shib Shankar Bose**, Afsana Sharmin**

Faculty, Students**, ****

Department of Computer Science & Engineering

Feni University Feni, Bangladesh, National Institute of Technology Silchar, India***

*Chittagong University of Engineering and Technology Chittagong, Bangladesh*****

Corresponding Author's email id: *mahmudbaharuddin@gmail.com*, Shibshankarbose72@gmail.com**, Rana99951@gmail.com****

DOI: - *http://doi.org/10.5281/zenodo. 3752498*

Abstract

In 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 color, and all the other 1-254 are the gray levels. 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 is different framework available to complete the task in parallel manner. Open CL is one of the frameworks for C/C++, Java and Python.

Keywords*: - Image processing, Histogram equalization, Image enhancement, Linear stretching, Parallel processing, Sequential processing GPU and CPU, Open CL.*

INTRODUCTION

Parallel processing is simultaneous use of multiple processors to execute certain 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. 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 the entire 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. Standard of OpenCL states a set of data types, data structures, and functions that augment C and C++. OpenCL ports have created by developers are available for Java and Python. However, standard only requires that OpenCL frameworks provide libraries structured language C and Object oriented C++. There are two major advantages of OpenCL , which are portability and parallel programming [1].

OpenCL includes aspects of concurrency, but one of its great advantages is that it enables parallel programming. Computational tasks assigned in parallel programming to multiple processing elements which is performed at the same time. These tasks are called kernels in OpenCL parlance [2]. 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. OpenCL provides full task-parallelism which is an useful advantage over many other parallel-programming toolsets that enables data-parallelism [3]. In a data-parallel system, each device receives the same instructions but operates on different sets of data.

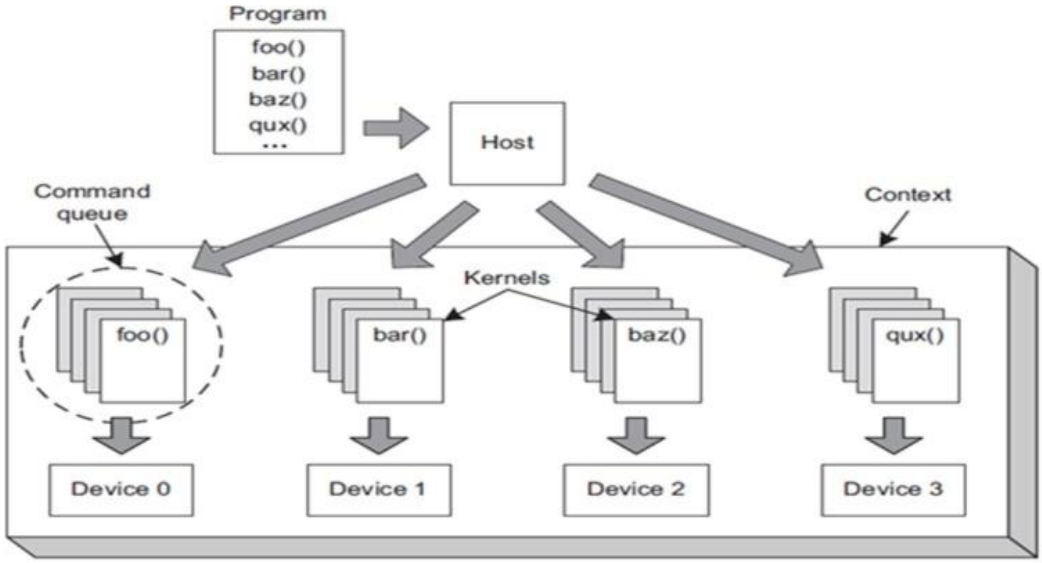

Figure 1: Kernel distribution in OpenCL tractable devices.

Parallel programming, vector processing, and probability make OpenCL strengthen than regular programming like C and C++, however greater power comes greater difficulties. In practical OpenCL application there has to create sets of different data structures and coordinate their task. It can be hard to keep everything straight forward.

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.

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. **See Figure: 2**

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.

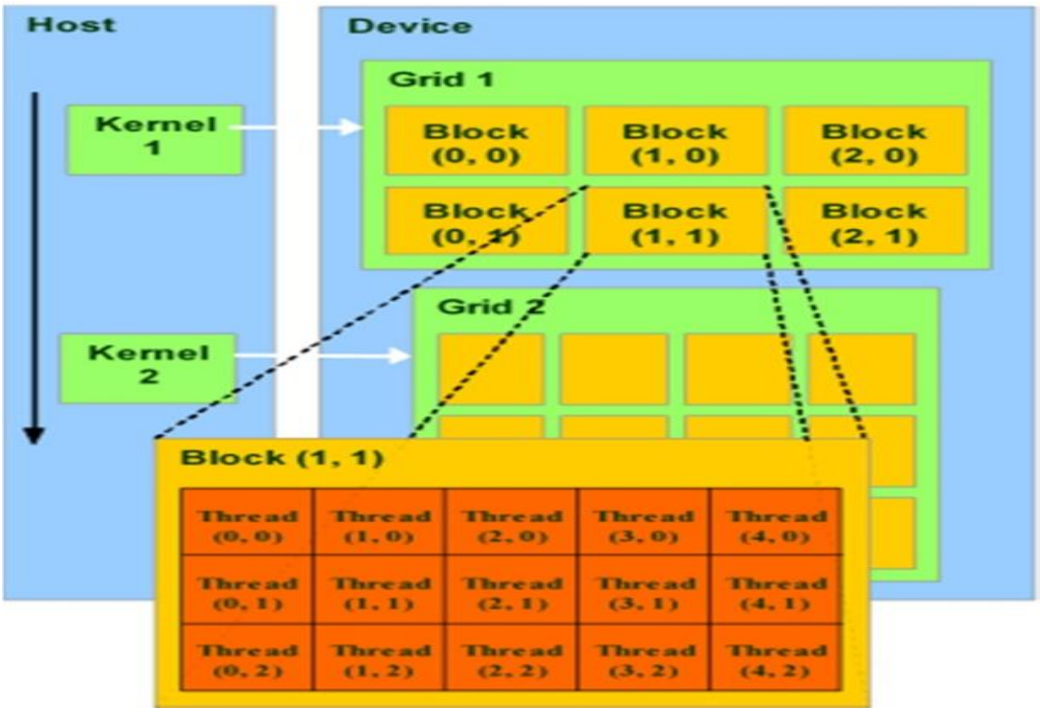

Figure 2: OpenCL programming model: showing CPU (host) to GPU (device) communication. Kernels are mapped to grids on the device, which execute blocks of threads concurrently

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.

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 [4]. 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.

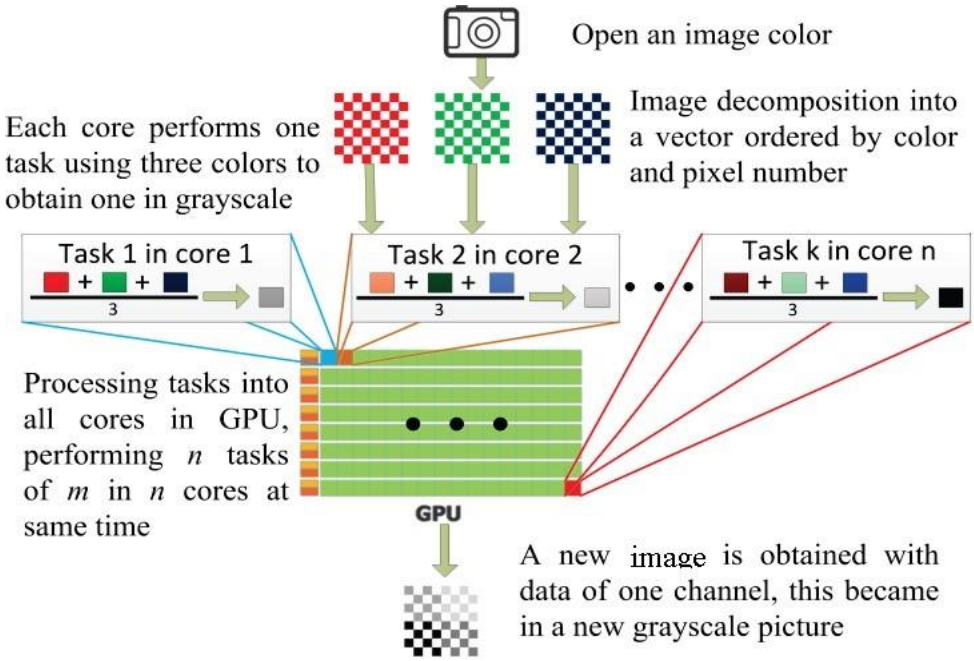

Figure 3: Figure showing how parallelism is used on GPU when images are used.

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.

OUR PROPOSED METHOD

A. Histogram equalization

Sometimes, we want to figure out detail of an image that can be tough to see with the naked eye. There have several techniques to improve the quality of an image in such a way named histogram equalization, which is commonly used to compare images made in entirely different circumstances. Example includes the comparisons of photographs with various angles of lighting, exposures and shadow casts [5]. 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. Such kind of histogram recommends that image has quality contrast and the details in the image can be observed flexibly. In histogram equalization, contrast of image is stretched so that image contrast is maximized by applying a gray level transform which tries to flatten the image histogram [6]. It turns out that the gray level transform that we are seeking is simply a scaled version of the original image's cumulative histogram.

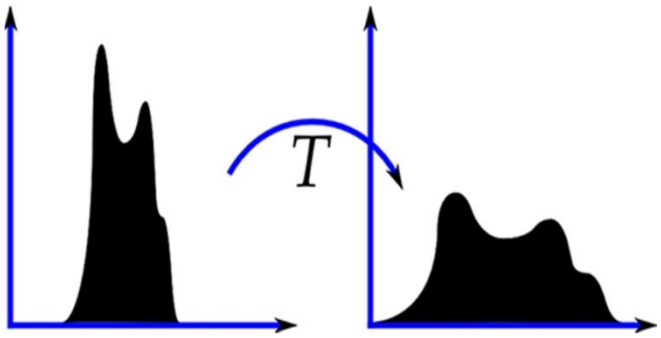

Figure 4: Histogram Equalization.

B. Histogram equalization algorithm

The main aim of histogram equalization is basically to expand cluttered frequencies over the length of the histogram that is more even. After that closed frequencies will be stretched out dramatically. It will result appearance of unseen details firmly of image that had little variation before.

A histogram equalization algorithm computes the number of times each frequency should found in the image and re-plot the image histogram properly. As the image data isn't garnered in an analogue formation which is usually not possible. Instead of this, the image data is actually stored digital manner and limited to n bits of image color depth. Which caused image cannot be quantified to meet our needs.

As we have to ensure the number of times particular color intensity is as closed as ideally equalized histogram, thus we have to perform re-quantification to the input image.

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 [7]. The algorithm computes the image intensity from 0 to N and shifts as many pixel frequencies into that position as long as this number of pixels. However, one thing should be keep in mind that the number of pixel must be less than or equal to a certain divider that stretching linear frequency of image.

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.

.

C. 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.

In linear stretching, output values in the output map obtained from the input values of a map.

First find out the upper and lower intensity value of input image. In histogram stretching, we actually stretched every intensity values into stretched range.

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. Followings are the formula of linear stretching [8]

$$\frac{inVal - inLow}{inHigh - inLow} = \frac{opVal - opLow}{opHigh - opLow} \quad (1)$$

$$opval = oplow + \frac{(inVal - inLow) * (opHigh - opLow)}{inHigh - inLow} \quad (2)$$

Where,

opVal = Value of pixel in output map

inHigh = Upper value of 'stretch from' x

inVal = Value of pixel in input map

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

inLow = Lower value of 'stretch from' range

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 which is shown in the figure 5.

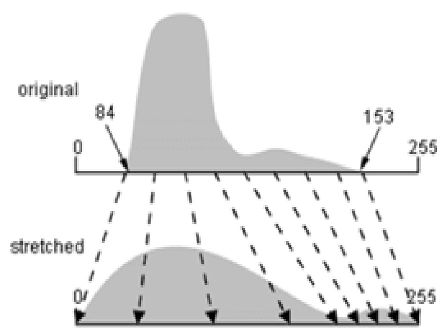

Figure 5: Histogram stretching

So the final usable formula can be derived

as

$$opVal = \frac{inVal - inLow}{inHigh - inLow} \quad (3)$$

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 = $255 * \frac{113-52}{26-52} = 101$

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.

D. Uses of histogram equalization

By using histogram equalization, the intensity of image would be better distributed by allowing the lower contrast area of image to higher contrast. This task is accomplished by spreading out the most frequent intensity values effectively. So it increases the contrast under exposed images and decreases over exposed images.

However, in case of RGB image, we can also perform the same procedure separately to the Red, Green and Blue colors.

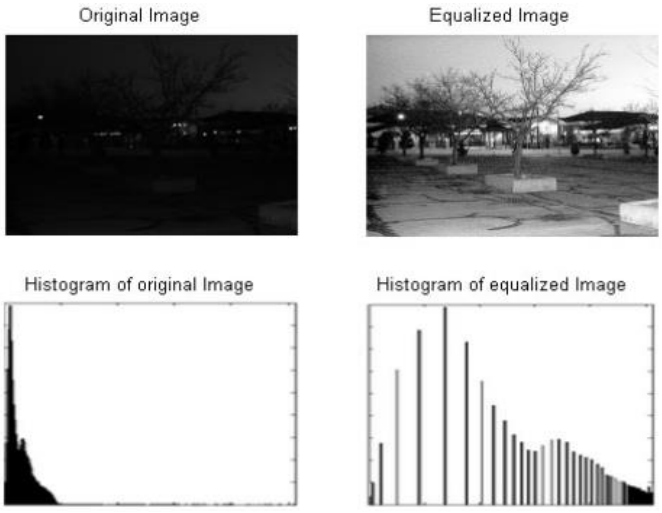

Figure 6: Histogram equalization of images with under exposure

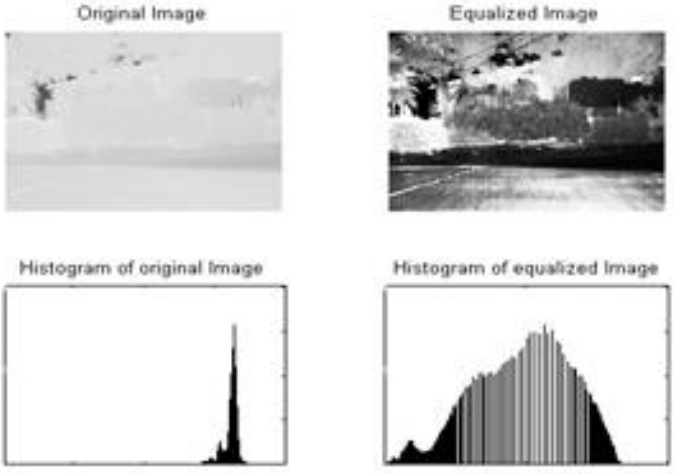

Figure: 7. Histogram equalization of images with over exposure

E. 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.

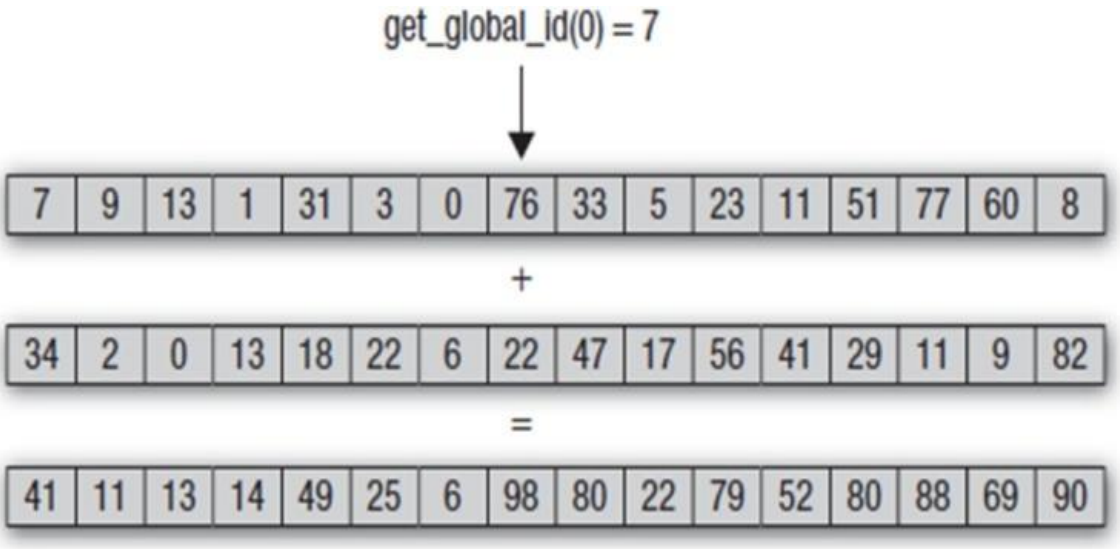

Figure 8: example of parallel addition of two vectors

For one batch of data, 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.

In order to 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

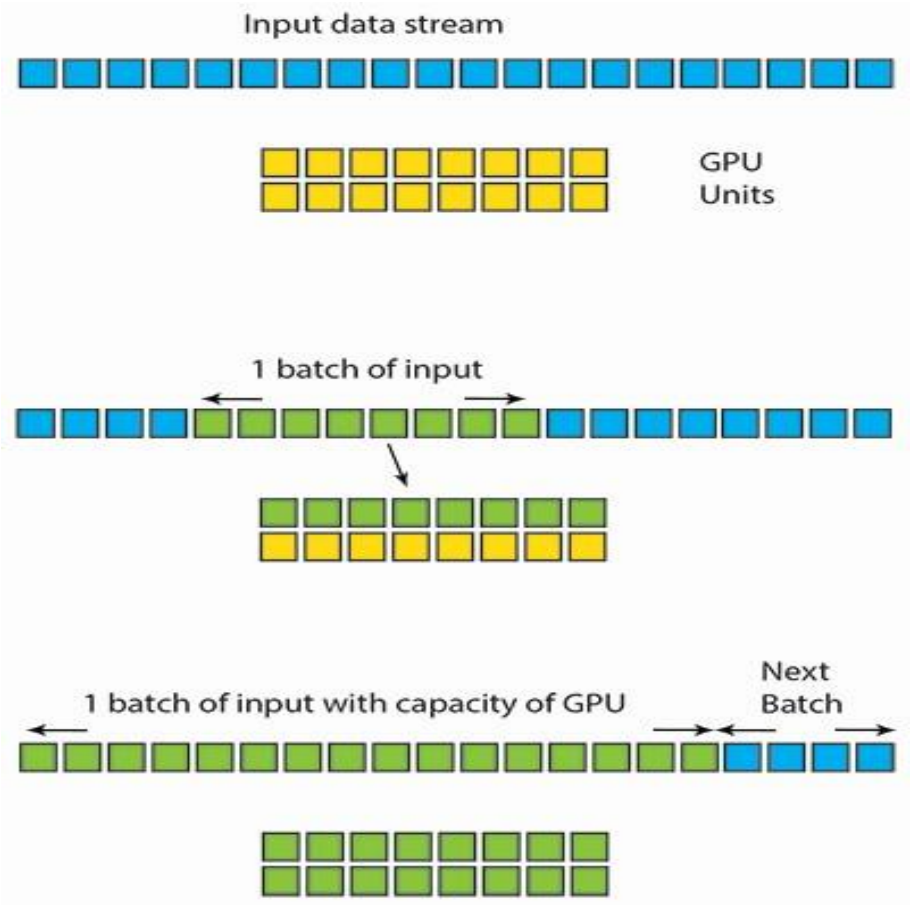

Figure 9: Passing the data on GPU.

RESULT AND DISCUSSION

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.

Statistics of input image matrix:

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

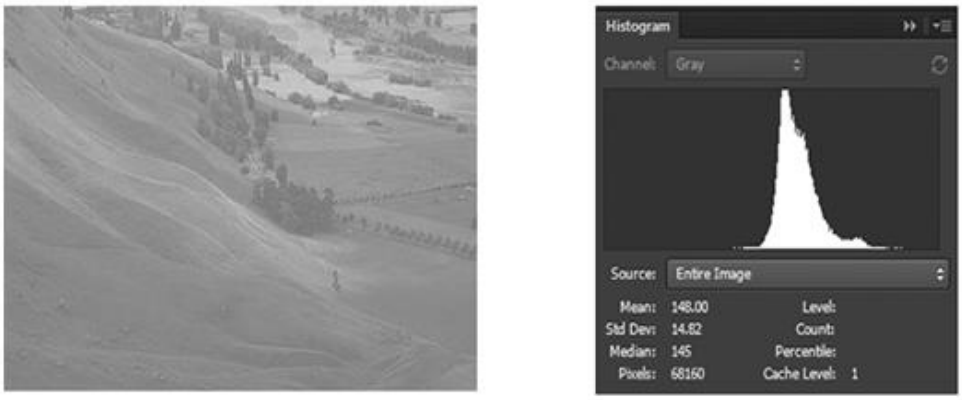

Figure 10: Input image

Statistics of output image matrix

Width = 320 pixels

Height = 213 pixels

Lowest intensity value = 0

Highest intensity value = 255

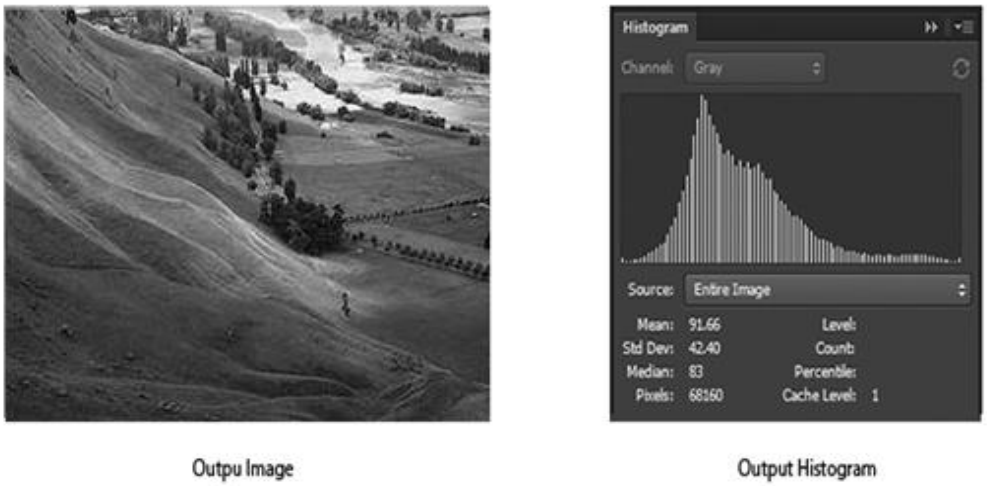

Figure 11: output image

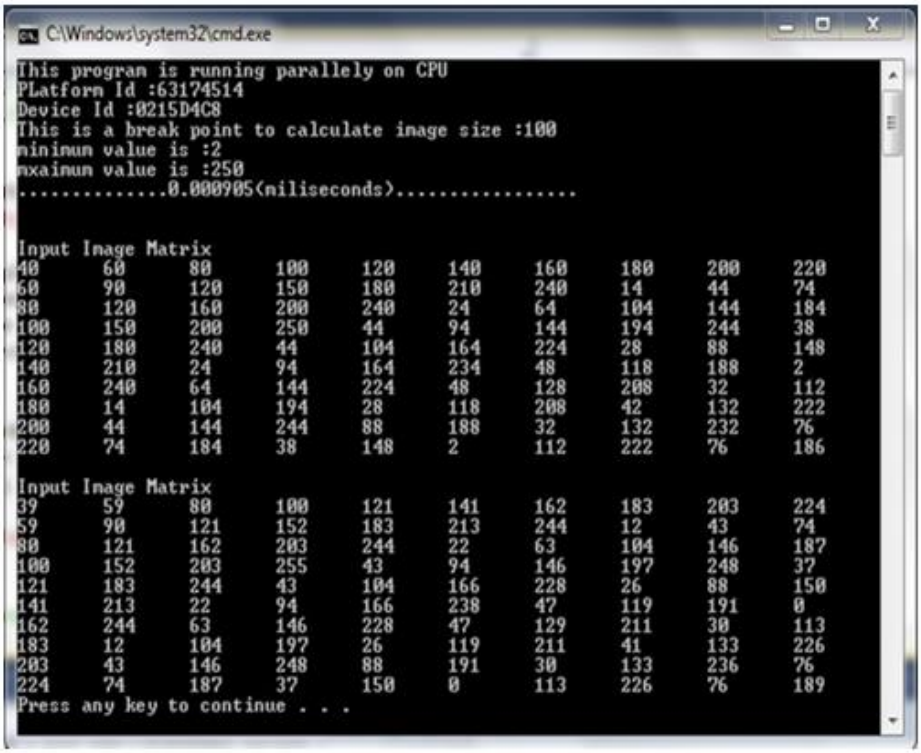

Figure 12: Snippet of output image

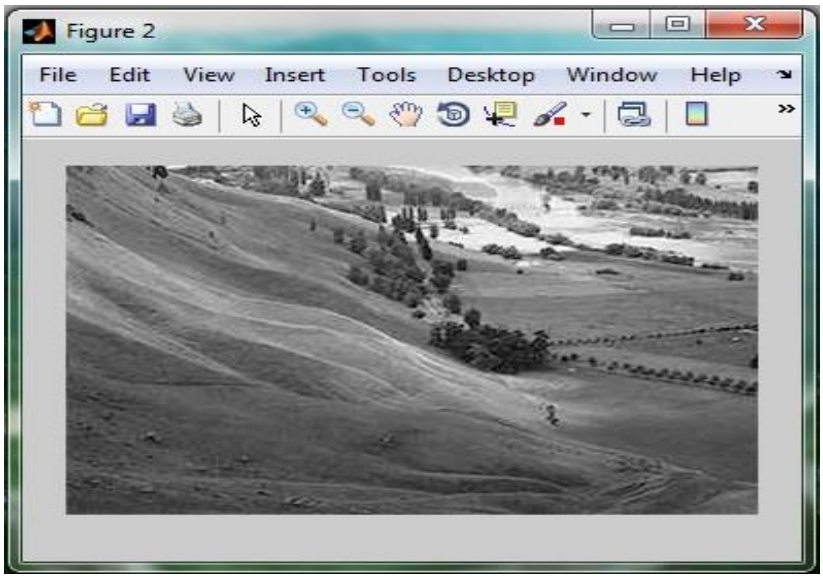

Figure 13: Final output image

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.

Timing Comparison with different processors and data are shown in the table I

Table 1: Timing Comparison

S. No.	Size of Input Matrix	Execution Timing on CPU	Execution Timing on GPU	Execution time on CPU
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

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.

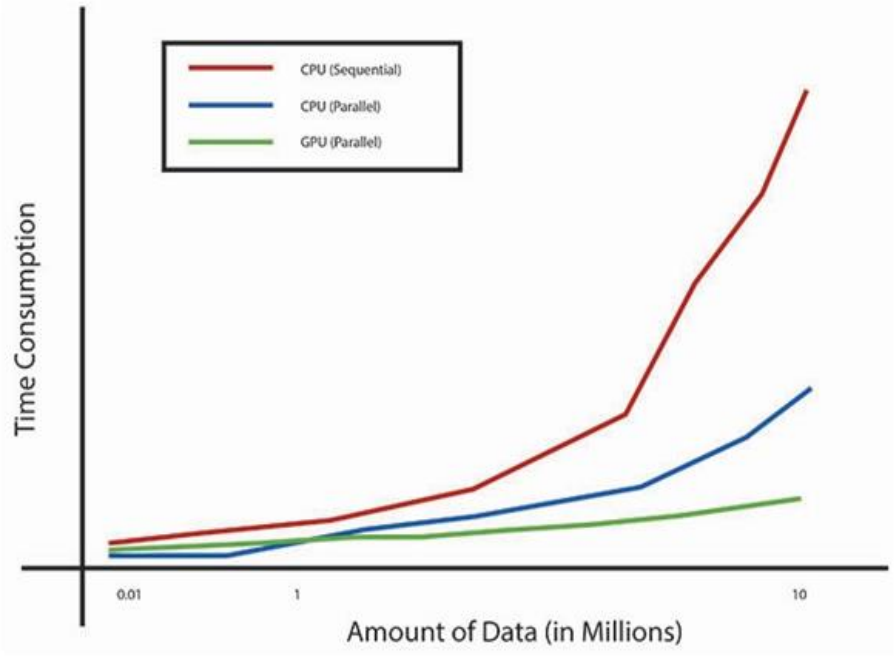

Figure 14: Comparison of time consumption v/s amount of data of different processors

CONCLUSION

This Parallel processing of image histogram dramatically reduces the time consumption over the high resolution images, when ran on GPUs in comparison to CPUs. The results show 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.

REFERENCES

I. "Introducing OpenCL", [Online]. Available: https://freecontent.manning.com/wp-content/uploads/introducing-opencl.pdf, Access date: 12/04/2019

II. " OPENCL IN ACTION: HOW TO ACCELERATE GRAPHICS AND COMPUTATIONS", [Online]. Available: https://epdf.pub/opencl-in-action-how-to-accelerate-graphics-and-computations13f18c9d972e9f8c9f0c1fa9fa77627b35050.html, Access date: 12/04/2019

III. https://epdf.pub/opencl-in-action-how-to-accelerate-graphics-and-computations13f18c9d972e9f8c9f0c1fa9fa77627b35050.html, Access date: 20/04/2019

IV. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, [Online], Access date: 11/06/2019

V. Ido Mildstein and Amir Fish, (June 1999), "Matlab interface to video camera and Face Detection", Stanford University, Access date: 27/04/2019

VI. https://www.scribd.com/document/45980831/Histogram-

Specification, Access date: 11/06/2019

VII. Ali Tarhini, “Face recognition: image processing". [Accessed: 14/06/2019].

VIII. http://spatial-analyst.net/ILWIS/htm/ilwisapp/stretch_algorithm.htm, Access date: 18/06/2019

Cite this Article as

Bahar Uddin Mahmud, Shib Shankar Bose, Afsana Sharmin (2020) “**Image Histogram Equalization using Parallel Processing (GPU computation) in Open CL to correct the Contrast of Image**s” Journal of Computer Aided Parallel Programming, 5 (1) 1-18
http://doi.org/10.5281/zenodo. 3752498