**Abstract:**

Historically digital image processing was applied on grayscale images because of the limitation with computational power in the early 80s. However, in the today's world power is not a major concern. But still gray scale images are widely used over color images in the field of image processing, security, computer vision etc.

The main motivation behind using the grayscale representations over the color images in image processing algorithms instead of operating on color images directly is that grayscale simplifies the algorithm and reduces computational requirements. Indeed, color may be of limited benefit in many applications and which may introduce unnecessary information and could increase the amount of training data required to achieve good performance.

Another important reason is that colors in most of the cases are meant for visual appeal of humans; but in many applications machines, are the main subject, like in computer vision. An application like object detection barely requires information at the edges of an image, which can be easily obtained in gray scale images too.

Gray scale images also help in reducing the processing time in many applications. It is often used when user wants the results as fast as possible. One of such application is where the application needs to match the currently clicked photo of the person with the millions of images present in the database which has the person’s photo with the profile details. for the identity of the person for security check.

In this project, we designed a parallel solution for the task of converting an image from the RGB scale to grayscale. Our implementation extracts the RGB color values from image pixels, calculates the intensity, generate data for creating the gray scale value using CUDA and produces the gray image as the output.

We have parallelized this problem in three ways:

1. Converting PNG to gray image using libpng library
2. Creating a new ppm format image and converting it into gray scale
3. Converting bmp to gray image using libpng library

Main implementation extracts the RGB color values from image pixels, calculates the intensity and creates the gray image.

**Introduction:**

In this project, we are taking RGB image as an input and proving its grayscale version as an output. With the help of the GPU and CUDA kernels grayscale value of each pixel is calculated in parallel.

Grayscale image is an image in which the value of each pixel is a single sample, that is, it carries only the intensity information. Grayscale images are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest. Therefore, images of this sort, are also known as black-and-white.

Color images are often built of several stacked color channels, each of them representing value levels of the given channel. For example, RGB images are composed of three independent channels for red, green and blue primary color components; CMYK images have four channels for cyan, magenta, yellow and black ink plates, etc.

Figure 1 shows the example of color channel splitting of a full RGB color image. The column at left shows the isolated color channels in natural colors, while at right there are their grayscale equivalences.

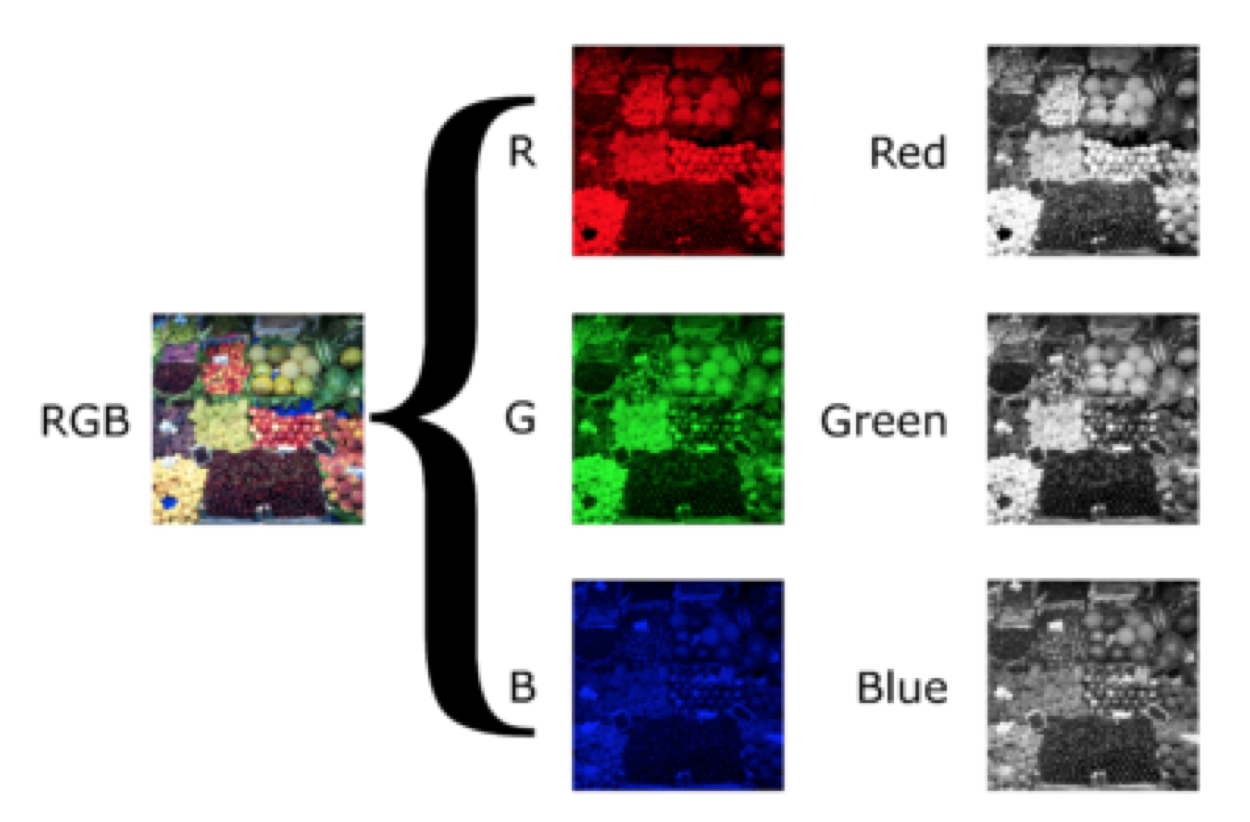


Figure 1: Color Channel Splitting of RGB Image

**Related Work:**

There are a number of methods to convert Color image to grayscale. Many of the methods for converting RGB images to grayscale employed dot products or weighted sums to map a three dimensional color space to a single dimension. Converting a color image to grayscale is a dimensionality reduction problem. In the course of our project, we referred [A Theory Based on Conversion of RGB image to Gray image International Journal of Computer Applications (0975 – 8887) Volume 7– No.2, September 2010] .

The paper proposed to split RGB image into R, G, and B component image and used the proposed transformation equation. Other projects that we found relevant are [Color-to-Grayscale: Does the Method Matter in Image Recognition] and [An Effective Grayscale Conversion with Applications to Image Enhancement] .

**Design:**

The Color to Gray is a global algorithm but no locality is needed to perform the calculation. Because of this, the image can be taken as a unidimensional signal. C++ has methods to read an image from the disk and convert its color representation from RGB to Intensity based on which gray colored images are created and has a header section which holds the information about the image, such as the height, the width and its number of pixels. After loading the input color image, its data member is loaded to the GPU and after the solving, the GPU gives a gray level data member that is copied back to a previously initialized gray image with same header of the color image. The pixel values are stored in an array of structs that holds the RGB channels values in each data member. The memory on the GPU is allocated after loading an image and determining its size in bytes. The input and output data are allocated to the Global Memory of the GPU. Once the memories are allocated and the input data is copied, we can pass to the kernel execution. The three different implementations are as follows:

1. **PNG to Gray Scale (What we broke)**

In this approach, we use a library libpng which helps in reading a png type of image file. We used png\_structp to read and create the files.

**Challenges/Limitation:** libpng was not installed on the lab machines. Hence, we were not able to check the implementation of parallel versions.

1. **BMP to Gray Scale**

To read the BMP type of image files, we created a function BMP where we used BMP file header(BITMAPINFOHEADER) to read the BMP image files. All the information related to BMP file like size of the image, color code information, bit size used etc. are saved in this header. RGB values are stored in different variables with in the header like biGammaRed for red, biGammaGreen for green and biGammaBlue for Blue. Then different color values for each pixel are read from these values using RGBQUAD structure. To store these values in a matrix, we multiplied the red values with 10^6 + green with 10^3 + blue values to reduce more and more interactions with global memory (This part is more explained in the Discussion section).

To convert the RGB pixel to Gray Scale pixels, we need to calculate the intensity using formula:

Intensity = 299f \* RedValues + .587f \* Greenvalue + .114f \* BlueValue



Figure 2: Code Snippet of the Intensity Calculation formula

For these conversions, we have implemented various versions of CUDA and CPP:

* **Sequential CPP**

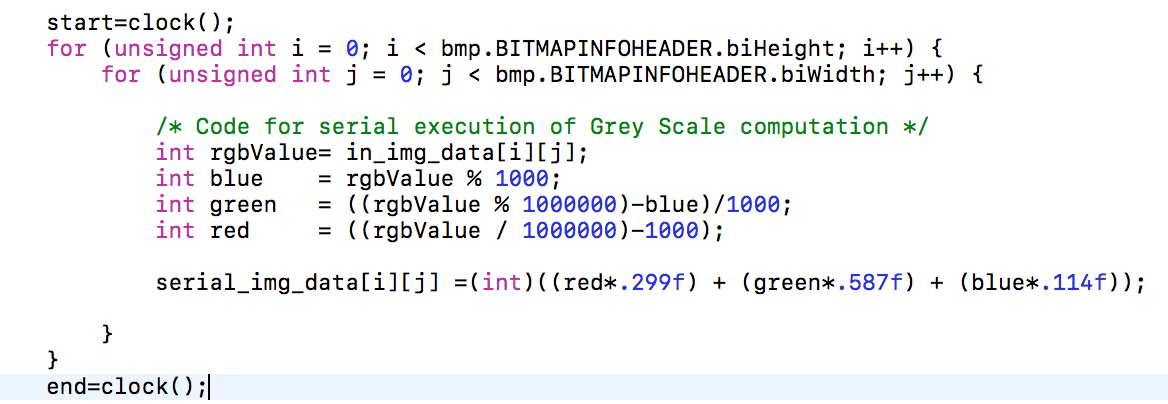


Figure 3: Code Snippet of Sequential program implemented in CPP

* **Cuda Naïve 2D implementation**

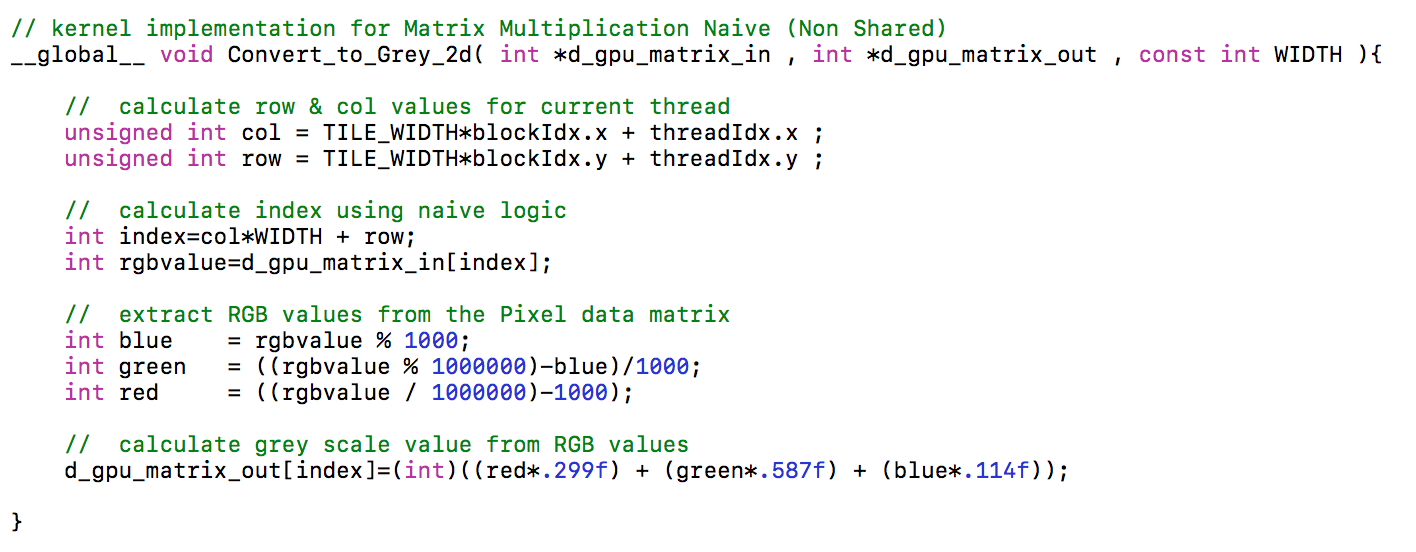


Figure 3: Code Snippet of Naïve 2D implementation in CUDA

* Cuda implementation using Shared memory

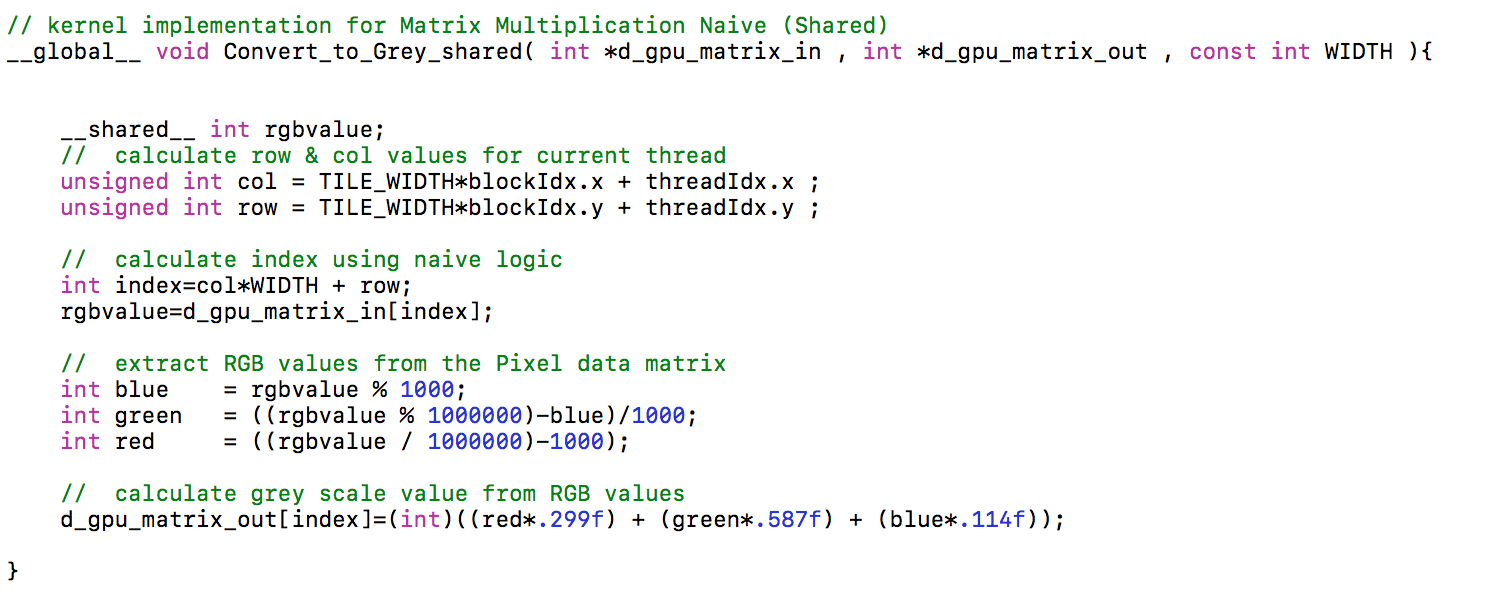


Figure 4: Code Snippet of Shared Memory implementation in CUDA

The RGB values stored in RGB info matrix are then decoded and stored in different variables respectively. The cuda kernels are conﬁgured in a way that each thread processes all pixels of the input image and write a single pixel on the output image. We changed the block and Grid dimensions to check the performance and execution of the program. The threads that exceed the image size must not write to output as they are out of the limits of the image, but must not be killed, because we need them to work in the data loading stage. The loading stage of the kernel execution consists of taking the input image pixels from the global memory and storing them in the shared memory block. The shared memory is on-chip and has very fast access time, but has very limited amount. So, we used shared memory in one implementation and it produced better results than other two.

After the intensity is calculated, Image is created using same BMP header file.

Challenges/Limitation :

Main challenge was to store the RGB values in a single matrix without using two additional matrices. This improved the performance drastically as the threads did not have to access/do data transfer operations from global memory again and again.

1. **PPM to Gray Scale**

In this implementation, instead of taking image input from user, we are giving user an option to create a colorful image of any size they want and file type PPM and then convert it into gray scale image. For this special case, we created an image with a vast range of 65025 color combinations and then converted this image to gray image to check the variations of color and performance of our cuda kernel.

**Challenges/Limitation:**

PPM type of image formats is not supported on every Operating System.

**Experiments and Result discussion:**

**System Specifications:**

GPU Specifications: Tesla M2090: 2.0

Global memory: 5331mb

Shared memory: 48kb

Constant memory: 64kb

Block registers: 32768

Warp size: 32

Threads per block: 1024

Max block dimensions: [ 1024, 1024, 64 ]

Max grid dimensions: [ 65535, 65535, 65535 ]

Number of multiprocessors: 16  
Number of CUDA cores: 512

Max mem pitch: [2147483647](tel:(214)%20748-3647)  
Memory Clock Rate (GHz): 1.848000  
Memory Bus Width (bits): 384  
Peak Memory Bandwidth (GB/s): 177.408000

Theoretical Peak Performance: 665 GFlops

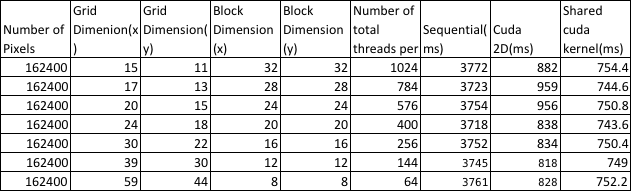


Table 1: Kernel Configurations for converting BMP type file format

**Observations:**

1. **Converting BMP to Grayscale:**

Graph 1: Performance comparison with different number of threads

Graph 2: Time Taken for conversion by different number of Threads/Blocks Configuration

**Discussion:**

* Total number of threads is defined as the product of threads per block and total number of blocks. As it can be seen from graph 1 and table 1, the performance changes with slight changes in number of threads. With increase in number of threads, performance gets better.
* This is due to the fact when there are more number of threads, each individual pixel is calculated by a single thread. Hence, there was no wait time for threads to finish instructions and then complete rest of the work
* From both graph 1 and 2, it is evident that cuda implementation of a program is much faster than sequential implementation. In this case, execution time for a basic cuda kernel is 4 times lesser than the execution time of sequential cpp implementation.
* Graph shows a drastic difference in performance of Naive cuda implementation and sequential implementation. Naive Cuda kernel is almost 4 times as good as sequential cpp code.
* We have taken block dimension (x,y) as multiples of 4 so there is always high occupancy and no threads in any warp are left unused. Due to this high warp occupancy, Cuda implementation performs 4 times better than CPU implementation.
* We also noticed that, performance of shared memory implementation as compared to the original implementation doesn’t affect the performance much. This is because , while calculating the intensity values in gray scale , there is no re use /sharing of variables between threads. So, the performance doesn’t get much affected.

Graph 3: Performance Measurement for Change in Number of Blocks/Grid

Graph 4: Time taken for Conversion by Different number of Blocks/Grid Configuration

**Discussion:**

* Similarly, in this case, we are plotting GLOPS and execution time against changing grid dimension.
* In this case, we are increasing the block size while the total number of thread remains constant.
* The performance (in GFlops) and time taken remains almost same, as we are not changing the total number of threads, in this case.
* As, represented in graph 4, the GFLOPS and execution time almost remains the same, with different values of number of blocks.

1. **Converting PPM to Grayscale:**

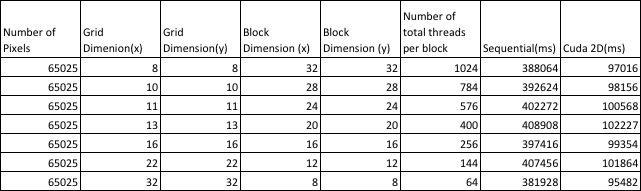


Table 2: Kernel Configurations for converting BMP type file format

Graph 5: Performance Measurements for change in Number of Threads/Block

Graph 6: Time Taken for Conversion by Different Number of Threads/Block configuration

**Discussion:**

* Total number of threads is defined as the product of threads per block and total number of blocks. As it can be seen from graph 5 and table 1, the performance is decreased if there is slight change in number of threads. With increase in number of threads, performance gets better.
* This is due to the fact when there are more number of threads, each individual pixel is calculated by a single thread. Hence, there was no wait time for threads to finish instructions and then complete rest of the work
* From both graph 5 and 6, it is evident that cuda implementation of a program is much faster than sequential implementation. In this case, execution time for a basic cuda kernel is 4 times lesser than the execution time of sequential cpp implementation.
* Graph shows a drastic difference in performance of Naive cuda implementation and sequential implementation. Naive Cuda kernel is almost 4 times as good as sequential cpp code.
* We have taken block dimension (x,y) as multiples of 4 so there is always high occupancy and no threads in any warp are left unused. Due to this high warp occupancy, Cuda implementation performs 4 times better than CPU implementation.

Graph 7: Performance Comparison with change in Number of Blocks/Grid

Graph 8: Time Taken for conversion by Different number of Blocks/Grid Configuration

**Discussion:**

* As we can infer from Table 1, when we are increasing number of blocks and decreasing number of threads.
* In this case, we are plotting GLOPS and execution time against changing grid dimension. In this case, we are increasing the block size, while the total number of thread remains constant.
* The performance (in GFlops) and time taken remains almost same, as we are not changing the total number of threads.

**Profiling:**

1. **On BMP file conversion:**

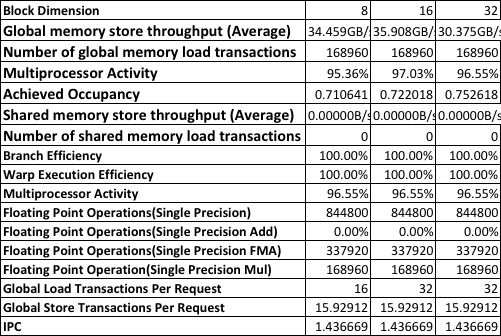


Table 3: Profiling values of BMP conversion file without implementing Shared memory CUDA

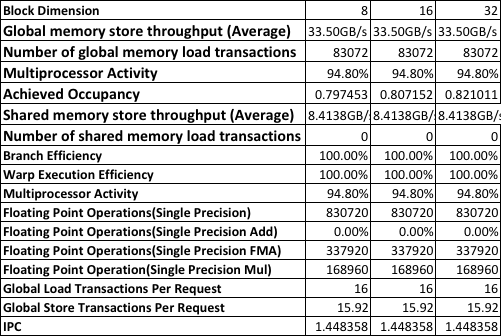


Table 4: Profiling values of BMP conversion file with implementing Shared memory CUDA

**Discussion:**

We applied profiling using nvprof on our second implementation, i.e Conversion of colour image to grayscale with and without using shared memory. We noticed only minimal effect in terms of performance and efficiency.

* The performance of the program improves slightly as each thread is working on individual pixels and has its own set of data to work on, which it loads into the shared memory. The data of one thread is of no use to other threads , as each thread is working on a single pixel.
* The global memory store throughput slightly increases , when using shared memory.The Shared memory store throughput was 0 initially with no shared memory implementation which increases to 8.4138 GB/s when shared memory was implemented.
* We have always taken block dimensions as multiples of 4 so that all the threads in a warp are used fully. Hence, we achieved the full warp occupancy. And, the Warp Execution Efficiency in the Profiling result remains 100% in both the cases.

1. **On PPM file conversion:**

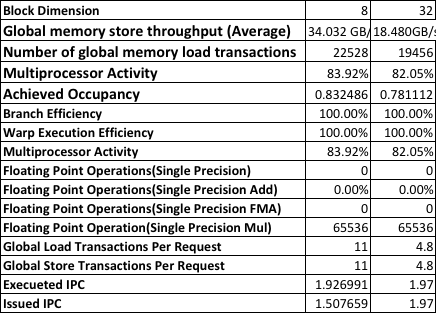


Table 5: Profiling values of PPM conversion file with change in Block Dimension (x,y)

**Discussion:**

* We applied profiling using nvprof on our implementation which uses PPM file format and converts color to Gray Scale method. We tried running profiling with two different values of block dimensions.
* With increase in dimension, we noticed that achieved occupancy decreases. This is because, we are creating a large number of threads with block dimension 32. Many of the threads remain unused, because of which the occupancy decreases.

**Conclusions:**

We tried implementing three different approaches to convert color image to grayscale .Each one of them has its own limitations and advantages.

* The first implementation using “pngx library” was the most versatile, as it gives programmers a lot of option to deal with the various pixel value matrices. The limitation of this approach was to have this “pngx library” installed before executing.
* In the second approach we are converting colored bmp image to grayscale. The user is free to select any bmp image.The advantage of using this approach is to use the c++ provided inbuilt structure “bitmapheader”, so there is no dependency on any external library.The limitation to this approach is handling and storing unnecessary values present in the structure, which is not required .This increases the complexity of the program and also leads to unnecessary memory consumption.
* The third approach , in which we use the basic grayfile format (PPM),gives the programmers the flexibility to create their own files with self defined pixel values.The limitation to this approach is the hassles of defining pixel values and not been able to work on any  given image.

From the above discussions, we could see that the problem of converting an image from RGB to gray in a sequential manner is very time consuming. Time consumed in the process increases as the image size increased. With the proposed parallel solution, same problem is solved in almost ¼ time with the help of GPU and CUDA compared to the sequential implementation. This proves that the parallel implementation is very efficient compared to sequential one.

Another observation from this project is that there is no significant performance gain with the implementation of shared memory for this particular problem. The reason behind minimal performance gain with the shared memory implementation compared to the global memory implementation is that there is no reuse of the data that each thread is loading into the memory.

**References:**

* <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.206.4410&rep=rep1&type=pdf>
* <http://cseweb.ucsd.edu/~ckanan/publications/Kanan_Cottrell_PloS_ONE_2012.pdf>
* <http://www.emo.org.tr/ekler/14281fad9e3d0cc_ek.pdf>
* <http://www.emo.org.tr/ekler/14281fad9e3d0cc_ek.pdf>
* <http://www.academia.edu/20415057/Parallel_Implementation_of_Grayscale_Conversion_in_Graphics_Hardware>
* <http://dl.acm.org.offcampus.lib.washington.edu/citation.cfm?doid=1667146.1667184>

