Intro to Parallel & Distributed Computing

Final Course Project-

Converting a Colored Image to Greyscale

Saurabh Singh
NetID- ss2716
Email: s.singh@rutgers.edu

Converting a Colored Image to Greyscale

a) The zip contains the following files:

1) main.out
2) main.cu
3) converter.cuh
4) helper.h
5) timeHelper.h
6) 10.png
7) 100.jpg
8) 1000.jpg
9) 5000.png

b) How to run:

1) Compile the code using the following command:
   nvcc `pkg-config --cflags opencv` main.cu `pkg-config --libs opencv` -o main.out
2) Execute the code with the below command
   ./main.out 100.jpg
   The code was built and tested on design.cs.rutgers.edu
3) The following three output files will be created in the same directory as the code is placed.
   i. greyscale_output_serial.png
   ii. greyscale_output_opencv.png
   iii. greyscale_output_cuda.png

c) Basic Requirement:

- The assignment tries to solve the problem of converting a colored image to a greyscale one.
- As the problem statement suggested the problem was first tried with a simple approach with setting the single channel value for the result as (R+G+B)/3.
- Initially a naïve method of coming up with the luma component just by taking the average of all the three channels is used.
- The result of this look converted to greyscale on the first observation but when observed carefully one can notice a few irregularities in the result. Although the results look convincing, they don't represent the real-world value. Our eyes perceive different channels differently so we need to use a better approach with
  Y' = 0.299*R + 0.587*G + 0.114*B.

d) Building on basic requirement- A parallel solution

- Since the problem involves manipulation in the pixel values and given pixels are internally stored as matrices, the problem lends itself quite naturally to parallel solution.
- Each pixel calculation being independent of the other pixel calculations also supports parallel solution

- In search of better performance, the following three implementations were designed
  i. Simple serial implementation
  ii. With Open CV
  iii. CUDA implementation

e) **Analysis:**

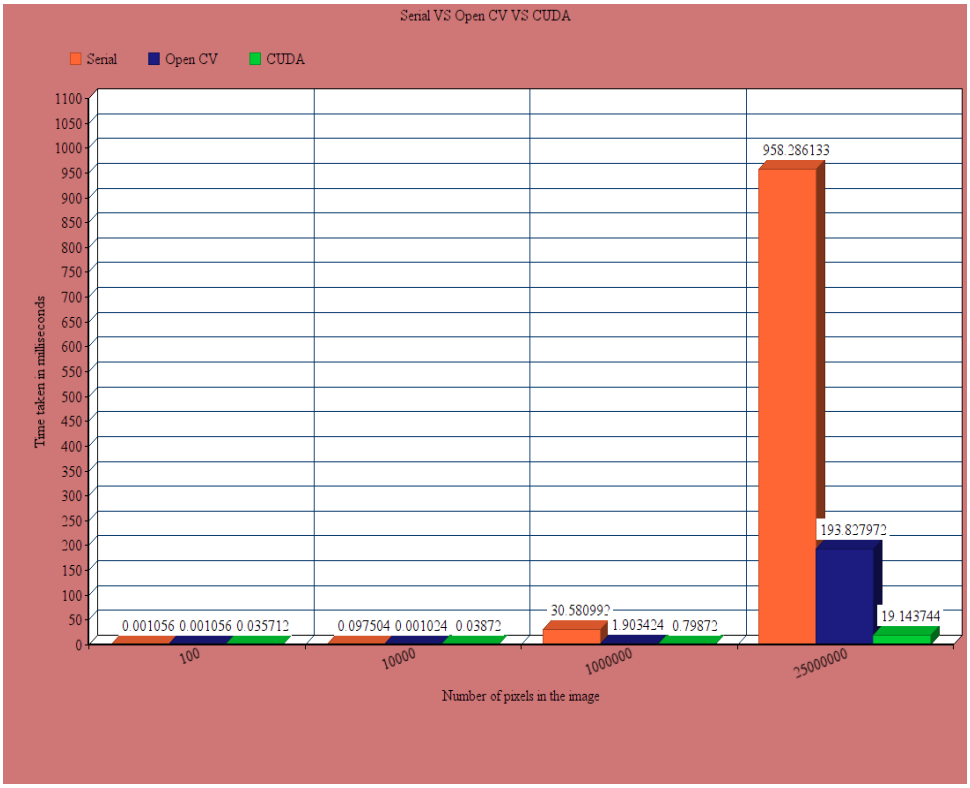
Following analysis were performed on the outputs:

1) compared the results of all three implementations for different image sizes and some interesting observations are as follows:

All time mentioned is in milliseconds.

Total number of pixels in the image	Serial	Open CV	CUDA
100	0.001056	0.001056	0.035712
10000	0.097504	0.001024	0.038720
1000000	30.580992	1.903424	0.798720
25000000	958.286133	193.827972	19.143744

The following bar graph depicts the results

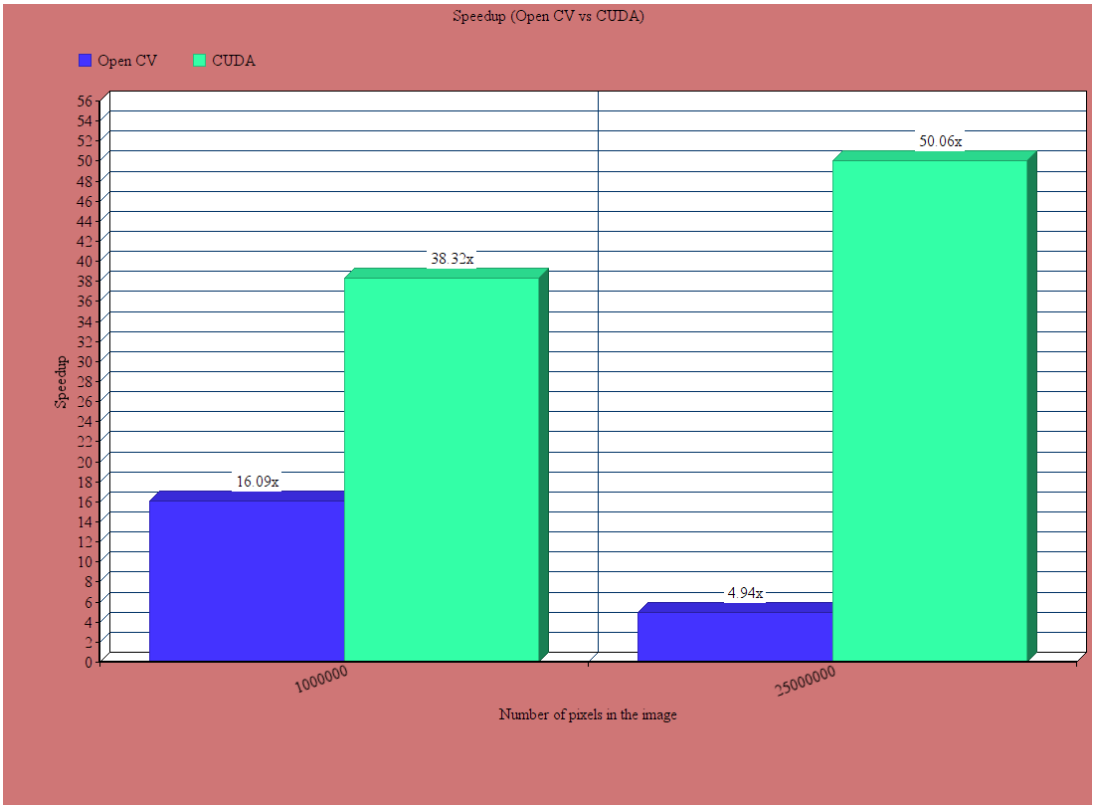


Observations:

- It is interesting to note that for very small work load (100-10000 pixels) CUDA is not the best choice and there is not enough workload to take advantage of CUDA parallelism. At this stage the overhead of computing the results on the device are grater. Serial implementation gives better results at low number of pixels.

- As the number of pixels go up, CUDA clearly takes over and performs much better the other two implementations

2) **<u>The following graph shows speedup achieved using CUDA for 1000000 and 25000000 pixel images</u>**

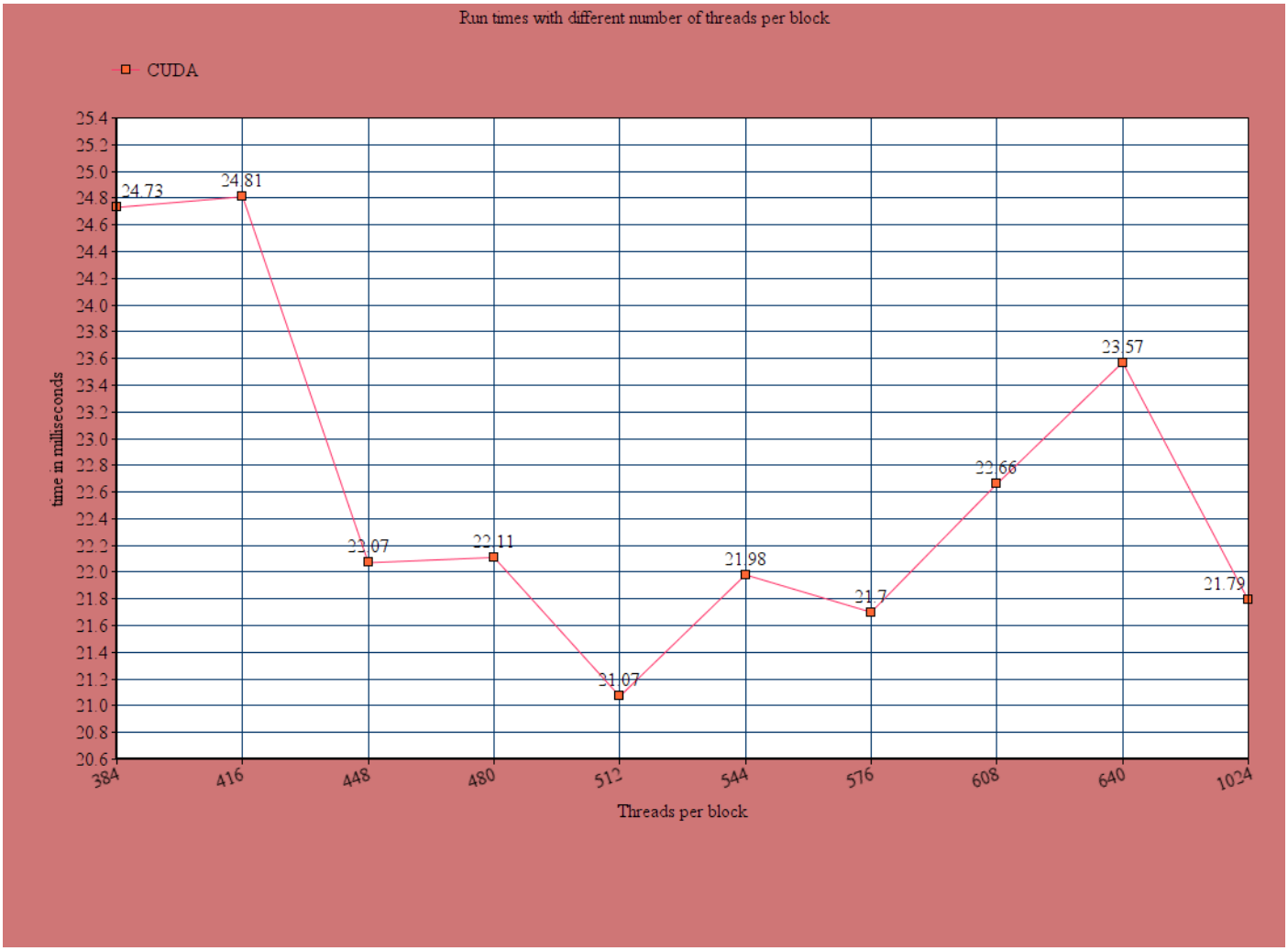


3) **<u>CUDA: Another analysis that was done was how many threads per block are optimum</u>**

- This was done keeping the number of pixels in the image constant. The highest definition image was chosen for this experiment with 25000000 pixels.
- We can have maximum of 65536 blocks per dimension of the grid.
- For simplicity one dimensional grid was picked. It is a common practice to keep the number of threads per block a multiple of wrap size (32 for most machines). This means we needed to have at least 384 threads per block. (25000000/384=65104).
- Starting with 384, the number of threads per block were increased and 10 iterations were done for each and average of those 10 run times was calculated to get the average time taken.
- This was done to find the sweet spot (optimum number of threads per block). The hardware limitations allowed maximum 1024 threads per block. The following table shows the results
  All times are in milliseconds

Threads per block	384	416	448	480	512	544	576	608	640	1024
Average time:	24.73	24.81	22.07	22.11	21.07	21.98	21.70	22.66	23.57	21.79

Following graph shows the plot of run times with different number of threads per block.



Observations:

- The optimum number of threads per block for this setup comes out to be 512.

----------The End----------