**Problem definition:**

This problem is automatic contrast enhancement on GPU. This is done by getting the minimum & maximum values of the image, subtracting minimum value from all images and then multiplying all pixels with 255.0/(maximum-minimum).

**Algorithm Description:**

*(the algorithm checks for CUDA errors after each CUDA operation)*

*There are more explanations on functions themselves.*

* Copy the image to device
* Go through all pixels and find minimum and maximum values with parallel reduction
  + In my implementation I first put smaller values to first half of the image, bigger values to second half of the image. Then I find minimum and maximum in a joint kernel call (named kernel1 & CPUfunc1).
    - In kernel1 the first pixel of the image becomes the minimum, the middle pixel (in this case 512\*512/2’th pixel) becomes the maximum.
* Go through all pixels and subtract minimum from them (named kernel2 & CPUfunc2).
* Go through all pixels and multiply them with 255.0/(maximum – minimum) (named kernel3 & CPUfunc3).
  + In my implementation I don’t need to use integer result scaling since I set my function so that it can take more than 1 Npp8u integer.
* Output the image to a .pgm file.
* Free the pointers.

**Benchmarking:**

Results are included in benchmark.txt

deviceQueryResult.txt is included.

One thing to note is that I am counting the time for GPU from the beginning of the first cudaMemCpy to the last cudaMemCpy, which unfortunately includes other calls such as std::cout, malloc etc.

**Pros-cons:**

* My GPU implementation uses slow operators such as ‘%’ 8 times and some simple operations are not optimized for performance. However, all these could be fixed with compiler optimizations, giving us a massive boost in performance.
* Using pitched arrays should be getting rid of bank conflicts.
* In kernel1 when there are less than 32(warp\_size) threads left, it keeps going without utilizing GPU’s full potential nor CPU’s potential (by sending the few elements to CPU for faster checking)
* There are 3 \_\_synchthreads() call in kernel1, perhaps the last call could be eliminated, giving us a small boost in performance.
* Using CPP templates could help performance.
* My GPU implementation closes complete warps first rather than closing individual(random) threads first.

**Discussion:**

What differs from the book/slides is that my GPU implementation seems slower than CPU implementation. This is due to use of multiple ‘%’ operators. Branching(s) in kernel1 is not a must since we can actually find the minimum of 2 values without comparing them, however perhaps finding the bigger value at the same time could be more expensive than just branching (if-else). Also, I was using more than 1024 threads per block when I started to code this, which took me more than 5-10 hours to figure out what was wrong.

**Environment:**

*DeviceQueryResult is included as a text file*

NVIDIA GeForce GTX 1060 3GB (compute capability 6.1)

Intel Core-I3 7100 @3.9GHz \w 8GB DDR4 RAM

Windows 10 Education Version 1809 (OS Build 17763.55)

Nvcc V10.0.130 compiler / Visual studio 2017 Community

**NOTE:** I didn’t had nppi.lib nor npp.lib, so I added all npp directories (cuda 10).