Assignment 02: GPU-based Computing

Assignment
You are requested to implement an image processing pipeline: the pipeline takes a color image as its input, converts it to grayscale, uses the image's histogram to enhance the contrast of the grayscale image and eventually returns a smoothed grayscale version of the input image. For simplicity and accuracy all operations are done in floating point. The program must be benchmarked on the NVIDIA GeForce GTX 750 TI GPU available on the SIMILDE machine. A brief description of the four phases is given below.

Phase 1: Converting a color image to grayscale
Our input images are RGB images; this means that every color is rendered adding together the three components representing Red, Green and Blue. The gray value of a pixel is given by weighting these three values and then summing them together. The formula is: gray = 0.3*R + 0.59*G + 0.11*B.

Phase 2: Histogram computation
The histogram measures how often a value of gray is used in an image. To compute the histogram, simply count the value of every pixel and increment the corresponding counter. There are 256 possible values of gray.

Phase 3: Contrast enhancement
The computed histogram is used in this phase to determine which are the darkest and lightest gray values actually used in an image - i.e., the lowest (min) and highest (max) gray values that have "scored" in the histogram above a certain threshold. Thus, pixels whose values are lower than min are set to black, pixels whose values are higher than max are set to white, and pixels whose values are inside the interval are scaled.

Phase 4: Smoothing
Smoothing is the process of removing noise from an image. To remove the noise, each point is replaced by a weighted average of its neighbors. This way, small-scale structures are removed from the image. We are using a triangular smoothing algorithm, i.e. the maximum weight is for the point in the middle and decreases linearly moving from the center. As an example, a 5-point triangular smooth filter in one dimension will use the following weights: 1, 2, 3, 2, 1. In this assignment, you will use a two-dimensional 5-point triangular smooth filter.

Accelerated application
A sequential version of the application is provided, for your convenience, in the directory “sequential”. Please read the code carefully, and try to understand it. A template for the parallel version is also provided, in the “cuda” directory. We recommend that you start from the template implementation that is provided. You need to parallelize and offload the previously described algorithms to the GPU. The "Kernel" comment in the code indicates the part that must be parallelized. There are no assumptions about the size of the input images, thus the code must be capable of running with color images of any size. The output must match the sequential version; a compare utility is provided to test for this. The output that you should verify is the final output image, named smooth.bmp. You are free to also save the intermediate images, e.g. for debugging, but do not include the time to write this images in the performance measurements. In the directory “images”, 16 different images are provided for testing. You can measure the total execution time of the application, the execution time of the four kernels and

the (introduced) memory transfer overheads. This way, you can compute the speedup over the sequential implementation, the achieved GFLOP/s and the utilization.

Compiling and running your application
Please use the provided Makefiles for compiling and running the codes.
Note: Cuda sdk is available in /opt/cuda. In order to use the sdk you need to source the file as:
source /opt/cuda/sourcemeCuda60
You can also put the lines in this file in your .bashrc file and you dont have to do it every time you log in or change terminal.

Report
Results are important but we are also interested in the procedure you followed to get those results and their analysis. In the report, you need to mention at least the following:

- Important points regarding implementation strategy. For instance, how did you tackle with the problem of reading/writing to single histogram value in histogram implementation?
- Problems/bottlenecks encountered. For instance how did you tackle the problem of reading/writing to same histogram values in Histogram computations?
- Optimizations performed, e.g. the use of shared memory, did it help or not and why did it help?
- Compiler options
- Comparison of the achieved speedup w.r.t sequential version
- Comparison of the achieved speedup and its analysis w.r.t peak computational power (in GFLOPS) and Memory Bandwidth (GB/sec)
- Interesting profiling and device utilization results

Speedup
The speed up results should be reported for image04, image09 and image15. Report the speedup as:

Overall Speedup = (Total Time on CPU) / (Total Time on GPU)

where

Total Time = (Execution Time) + (Communication Time)

Furthermore,

Kernel Speedup = (Kernel Execution Time on CPU) / (Kernel Execution Time on GPU)

Do not include the printing/debugging etc in timing calculations.

Code
Submit a single archive containing your working solution. Check the output images after each step and compare to see if they are correct. Clean the code and do not include images in this archive (as they are too big). Timing calculations and meaningful printing of results should be inside code so that we can verify your reported results by simply running it (without the need to write the timing code ourselves). Put the comments in your code to explain what are you doing, so that we know you understand it.

Optional
You can also run the solutions on other Nvidia cards in case you have access (e.g. the one in your laptop). The cuda code is portable, but is the performance portable?

Best of Luck and Enjoy GPU programming...