Accelerators and Accelerated Systems - 046274
Spring 2019

Homework 1

Submission via moodle only.
Due: 16/5/2019 23:55
Make sure to submit a zip/tar.gz archive with 2 files: ex1.cu, ex1.pdf.
Archive name should be IDs of the students, separated with underscore
(e.g. 123456789_123571113.tar.gz)

_____

In this assignment we will implement histogram equalization for gray scale images. This method stretches the color range of an image in order to improve its contrast.

Below you can see a pair of images. On the left, the original image, and on the right, the image after histogram equalization. (source: https://en.wikipedia.org/wiki/Histogram_equalization)

Histogram equalization works by first creating a histogram of the gray levels in the image, then using the histogram to create a map which maps each value of these levels to a new value, and finally, using this map to create the new image.

We will implement histogram equalization on MxN gray scale images, each represented as a `unsigned char` array of length M*N, with values in the range [0, 255], where 0 means black, and 255 means white.

We perform histogram equalization as follows:

1. Create a histogram *h*: an array of length 256. h[v] is the number of pixels which have the value v.
2. Create the cumulative distribution function (*CDF*) from the histogram.
   CDF[v] = h[0] + h[1] + … + h[v]
3. Calculation $CDF_{min} = min\{x \in CDF \mid x \neq 0\}$
4. Create a map *m* from old gray level to new gray level. m[v] is the new value of pixels which originally had the value v.
   m is computed as follows:

$$m(v) = floor\left(255 \times \frac{cdf(v) - cdf_{min}}{M \times N - cdf_{min}}\right)$$

5. Compute the new image:
$$new[i] = m[original[i]]$$

for each pixel i.

Submission Instructions:
You will be submitting an archive (id1_id2.tar.gz or id1_id2.zip) with 2 files:
1. ex1.cu:
   1.1. Contains your implementation.
   1.2. Make sure to check for errors, especially of CUDA API calls. A CUDA_CHECK() macro is provided for your convenience.
   1.3. When measuring times, you should measure the time it takes for the memory movements (memcpy) and the computations. Memory allocations (cudaMalloc, cudaHostAlloc) should **not** be counted.
   1.4. Your code will be compiled using: "nvcc -O3 hw1.cu -o hw1". Make sure it compiles **without warnings** and runs correctly before submitting.
   1.5. Free all memory and release resources once you have finished using them.
   1.6. A skeleton ex1.cu file is provided along with this assignment. Use it as a starting point for your code.
2. ex1.pdf:
   2.1. A report with answers to the questions in this assignment.
   2.2. Please submit in pdf format. **Not .doc** or any other format.
   2.3. No need to be too verbose. Short answers are ok as long as they are complete.

Tasks:

1. **Knowing the system:**
    1.1. In the report: Report the CUDA version on your machine (use: nvcc --version)
    1.2. In the report: Report the GPU Name (use: nvidia-smi command)
    1.3. Copy the directoy "/usr/local/cuda/samples/" to your home directory. Then go to the subdirectory 1_Utilities/deviceQuery. Compile it (make) and run ./deviceQuery. Should show information about your GPU. Examine the output proudly and report the number of SMs (Multiprocessors) int the report.

2. **Implement device functions:**
    2.1. Implement the device function
    ```
    __device__ int array_min_positive(int *arr, int len)
    ```
    2.2. Implement the device function
    ```
    __device__ int prefix_sum(int *arr, int len)
    ```

3. **Implement a task serial version:**
    3.1. Implement a kernel which takes an image as an array from global memory, and returns the new equalized image to another location in global memory. Invoke this kernel on the input images in a for loop.
    3.2. You will want to use atomicAdd to compute the histogram. Refer to "http://docs.nvidia.com/cuda" to learn how to use it.
    3.3. In the report, explain why atomicAdd is required.
    3.4. Use the following pattern: memcpy image from CPU to GPU, invoke kernel, memcpy output image from GPU to CPU.
    3.5. Use a reasonable number of threads when you invoke the kernel. Explain your choice in the report.
    3.6. In the report: Report the total run time and the throughput (images / sec). memcpy-s should be included in this measurement.
    3.7. In the report: Use NVIDIA's visual profiler to examine the execution diagram of your code. Attach a clear screenshot to the report showing at least 2 kernels with their memory movements.
    3.8. In the report: Choose one of the memcpy's from CPU to GPU, and report its time (as appears in NVIDIA's visual profiler)

4. **Implement a bulk synchronous version:**
In the previous task, we utilized a small fraction of the GPU at a given time. Now we will feed the GPU with all the tasks at once
    4.1. Implement a version of the kernel which supports being invoked with an array of images and mulitple threablocks, where the block index determines which image each threadblock will handle. Alternatively, modify the kernel from (3) to support multiple threadblocks.
    4.2. Invoke the kernel on all input images at once, with number of threadblocks == number of input images.
    4.3. Use this pattern: memcpy all input images from CPU to GPU, invoke 1 kernel on all images, memcpy all output images from GPU to CPU.
    4.4. In the report: Report the execution time, and speedup compared to (3).
    4.5. In the report: Attach a clear screenshot of the execution diagram from NVIDIA's visual profiler.
    4.6. In the report: Check the time CPU-to-GPU memcpy in NVIDIA's visual profiler. Compare it it to memcpy time you measured in (2). Does the time grow linearly with size of the data being copied?