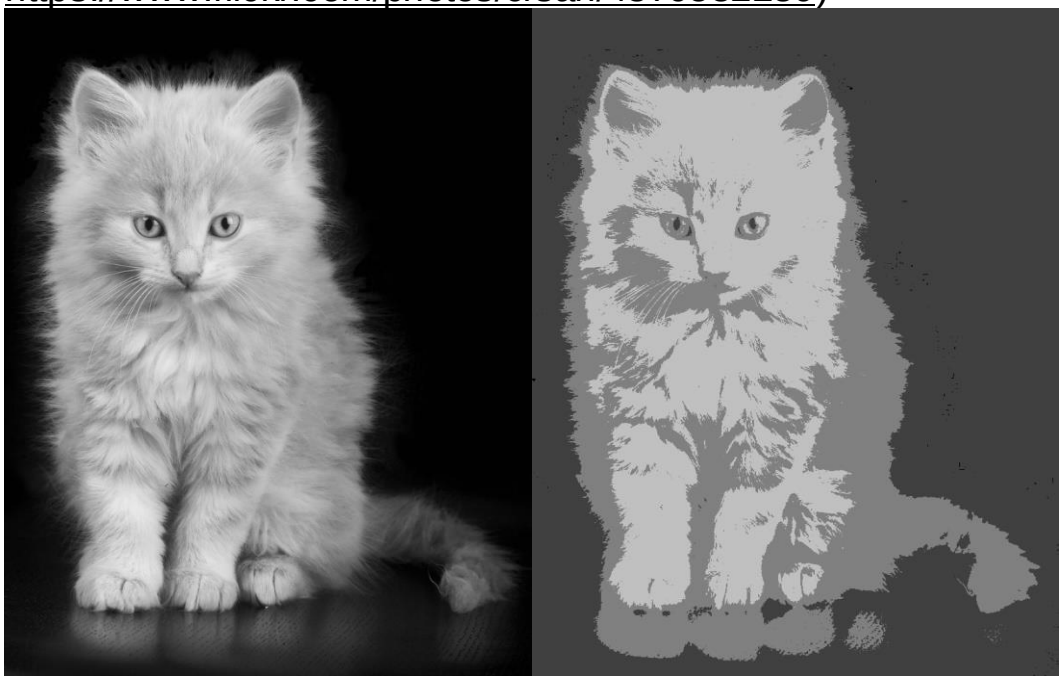
Homework 1

Submission via Moodle only.
Due: 30/4/2020, 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 image quantization for grayscale images. This method reduces the number of colors used in an image.

Below you can see a pair of images. On the left, the original image, and on the right, the image after quantization to 4 colors. (source: https://www.flickr.com/photos/crsan/4379532286)



The algorithm we will implement 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 quantization on MxN grayscale 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 quantization into L gray levels 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. 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:
   $$m(v) = \frac{256}{L}\left\lfloor L \cdot \frac{cdf(v)}{M \cdot N} \right\rfloor$$
4. Compute the new image. For each pixel i:
   $$new[i] = m[original[i]]$$

Homework package

The archive includes the following files:

Filename	Description
`homework1.pdf`	This file.
`ex1.cu`	A template for you to implement the exercise and submit. **This is the only file you need to edit**.
`ex1-cpu.cu`	A CPU implementation of the algorithm that is used to check the results.
`main.cu`	A test harness that runs your algorithm on random images, and compares the result against the CPU implementation above, as well as measures performance.
`image.cu`	A test program that runs the CPU implementation against an image file and produces an output image file, for your curiosity, by running: `./image <image file>`
`cat.jpg`	A cat shown above.
`Makefile`	Allows building the exercise `ex1` and the graphical test application `image` using `make ex1` and `make image` respectively.

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. The existing test harness in main.cu already does that.
   1.4. Your code will be compiled using: "make ex1". 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.
1.7. Do not submit other header files and .cu files provided in the homework archive, and do not modify them.

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.

1.4. In the report: Examine the output proudly and report the number of SMs (Multiprocessors) in the report.

2. Implement device functions:

2.1. Implement the device function `prefix_sum` to calculate a prefix sum of the given array in-place:

```
__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 processed image to another location in global memory. Invoke this kernel on the input images in a for loop.

```
__global__ void process_image_kernel(uchar *all_in, uchar *all_out)
```

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. Define the state needed for the task serial in the `task_serial_context` struct.

3.5. Allocate necessary resources (e.g. GPU memory) in the `task_serial_init` function, and release it in the `task_serial_free` function.

3.6. Implement the CPU-side processing in the `task_serial_process` function. Use the following pattern: memcpy image from CPU to GPU, invoke kernel, memcpy output image from GPU to CPU.

3.7. Use a reasonable number of threads when you invoke the kernel. Explain your choice in the report.

3.8. In the report: Report the total run time and the throughput (images / sec). memcpy-s should be included in this measurement.

3.9. 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.

3.10. 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. Define the state needed for the bulk synchronous version in the `gpu_bulk_context` struct.

4.3. Allocate necessary resources (e.g. GPU memory) in the `gpu_bulk_init` function and release it in the `gpu_bulk_free` function.

4.4. Invoke the kernel on all input images at once, with number of threadblocks == number of input images in the `gpu_bulk_process` function. 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.

4.5. In the report: Report the execution time, and speedup compared to (3).

4.6. In the report: Attach a clear screenshot of the execution diagram from NVIDIA's visual profiler.

4.7. 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?