Homework 1

Submission via Moodle only.

Due: 25/4/2022, 23:55

Make sure to submit a zip/tar.gz archive with 2 files: ex1.cu, ex1.pdf. The archive name should be the IDs of the students, separated with an underscore (e.g. 123456789_123571113.tar.gz)

Please read this entire document before you begin coding or answering questions.

Assignment goal

In this assignment, we will implement histogram equalization for grayscale images. This method is used to improve contrast in images. Specifically, we will implement adaptive histogram equalization. You can read more about it on Wikipedia: https://en.wikipedia.org/wiki/Adaptive_histogram_equalization.

Below you can see a pair of images. On the left is the original image, and on the right is the image after adaptive histogram equalization.

(source: https://www.flickr.com/photos/woolamaloo_gazette/4170378410)

The algorithm

This is the algorithm that we will implement:

1) Dividing the image into square tiles.
2) For each tile:
   a) Create a histogram of the gray levels in the tile.
   b) Use the histogram to create a map, which maps each gray level value to a new gray level value.
3) For each pixel in the image:
   a) Find the four closest tiles' centers
   b) Use the mapping of these four tiles to get four new gray levels.
   c) Use interpolation to determine the pixel's new value.

You can see more details on Wikipedia:
https://en.wikipedia.org/wiki/Adaptive_histogram_equalization#Efficient_computation_by_interpolation

We make the following simplifying assumptions:

- The tiles are square, with size $T \times T$
  - $T \geq 64$
  - $T = 2^k, k \in \mathbb{N}$
- the images are square, with size $N \times N$, where N is a multiply of T.
- Each image is represented as an unsigned char array of length $N \cdot N$, with values in the range [0, 255], where 0 means black, and 255 means white.

We calculate the map for each tile as follows:

1) Create a histogram h: an array of length 256. $h[v]$ is the number of pixels that have the value v.
2) Create the cumulative distribution function (CDF) from the histogram. $CDF[v] = h[0] + h[1] + \cdots + h[v]$
3) Create a map m from the old gray levels to the new gray levels. $m[v]$ is the new value of pixels that originally had the value v. m is computed as follows:

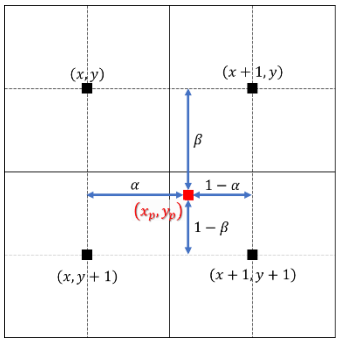
$$m[v] = \left\lfloor \frac{CDF[v]}{T \cdot T} \cdot 255 \right\rfloor$$

We perform interpolation as follows:

- **NOTE: the implementation of the interpolation is supplied for you.**
  - **Just call interpolate_device() after you prepare the maps.**
  - Signature:

```
/**
 * Perform interpolation on a single image
 *
 * @param maps 3D array ([TILES_COUNT][TILES_COUNT][256])
 *             of the tiles' maps , in global memory.
 * @param in_img single input image, in global memory.
 * @param out_img single output buffer, in global memory.
 */
__device__
void interpolate_device (uchar* maps ,uchar *in_img,
uchar* out_img);
```

- The interpolation scheme is presented below for completeness:



1. Given pixel (x_p, y_p) with value $v = img[y_p][x_p]$
2. Find the four closest tile's centers: $(x, y), (x + 1, y), (x, y + 1), (x + 1, y + 1)$
3. Use the maps of these tiles, to find four new gray-level values:
   $v'_{(x,y)} = map_{(x,y)}[v]$
   $v'_{(x+1,y)} = map_{(x+1,y)}[v]$
   $v'_{(x,y+1)} = map_{(x,y+1)}[v]$
   $v'_{(x+1,y+1)} = map_{(x+1,y+1)}[v]$
4. Calculate the normalized distance of the pixel (x_p, y_p) from the centers:
   $\alpha = \frac{x_p - x}{T}$
   $\beta = \frac{y_p - y}{T}$
5. Finally, calculate the new pixel value:
   $$v' = (1-\alpha)(1-\beta)v'_{(x,y)} + \alpha(1-\beta)v'_{(x+1,y)} + (1-\alpha)\beta v'_{(x,y+1)} + \alpha\beta v'_{(x+1,y+1)}$$

Homework package

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, compares the result against the CPU implementation above, and measures performance.
image.cu	A test program that runs the CPU implementation against a square image file and produces an output image file (result.png), for your curiosity, by running: ./image <image file>
city.jpg	The city 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:
   a) Contains your implementation.
   b) Make sure to check for errors, especially of CUDA API calls. A CUDA_CHECK() macro is provided for your convenience.
   c) 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.
   d) Your code will be compiled using: “make ex1”. Make sure it compiles without warnings and runs correctly before submitting.
   e) Free all memory and release resources once you have finished using them.
   f) A skeleton ex1.cu file is provided along with this assignment. Use it as a starting point for your code.
   g) Do not add other header files or .cu files. Do not modify other files.
2) ex1.pdf:
   a) A report with answers to the questions in this assignment.
   b) Please submit in pdf format. Not .doc or any other format.
   c) No need to be too verbose. Short answers are OK as long as they are complete.

Tasks

1) **Knowing the system:**
   a) In the report: Report the CUDA version on your machine (use: nvcc --version)
   b) In the report: Report the GPU Name (use: nvidia-smi command)
   c) Copy the directory "/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.
   d) In the report: Examine the output proudly and report the number of SMs (Multiprocessors) in the report.
2) **Implement device functions:**
   a) 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:**
   a) Implement a kernel that takes an image as an array from global memory and returns the processed image to another location in global memory.

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

   - the number of threads should be a multiple of the tile width: $\#threads = k \cdot T, k \in \mathbb{N}$.
   - You will Invoke this kernel in a for loop, on a single input image at a time, with a single thread block.
   - You will want to use atomicAdd to compute the histogram. Refer to "http://docs.nvidia.com/cuda" to learn how to use it.

   b) In the report, explain why atomicAdd is required.
   c) In the report, explain why regardless of the number of threads, your global memory accesses are coalesced.
   d) Define the state needed for the task serial in the task_serial_context struct.
   e) Allocate necessary resources (e.g. GPU memory) in the task_serial_init function, and release them in the task_serial_free function.
   f) 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.
   g) Use a reasonable number of threads when you invoke the kernel. Explain your choice in the report.

h) In the report: Report the total run time and the throughput (images / sec). memcpy-s should be included in this measurement.
i) In the report: Use NVIDIA Nsight Systems to examine the execution diagram of your code. Attach a clear screenshot to the report showing at least two kernels with their memory movements.
   - (install NVIDIA Nsight Systems on your computer and run the profiling remotely)
   - You will need to zoom-in a loot.
j) In the report: Choose one of the memcpy's from CPU to GPU, and report its time (as appears in NVIDIA Nsight Systems).

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

a) feed the GPU with all the tasks at once
b) Implement a version of the kernel which supports being invoked with an array of images and multiple threablocks, where the block index determines which image each threadblock will handle. Alternatively, modify the kernel from (3) to support multiple threadblocks.
c) Define the state needed for the bulk synchronous version in the gpu_bulk_context struct.
d) Allocate necessary resources (e.g. GPU memory) in the gpu_bulk_init function and release them in the gpu_bulk_free function.
e) 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 one kernel on all images, memcpy all output images from GPU to CPU.
f) In the report: Report the execution time and speedup compared to (3.h).
g) In the report: Attach a clear screenshot of the execution diagram from NVIDIA Nsight Systems.
h) In the report: Check the time CPU-to-GPU memcpy in NVIDIA Nsight Systems. Compare it to memcpy time you measured in (3.j). Does the time grow linearly with the size of the data being copied?