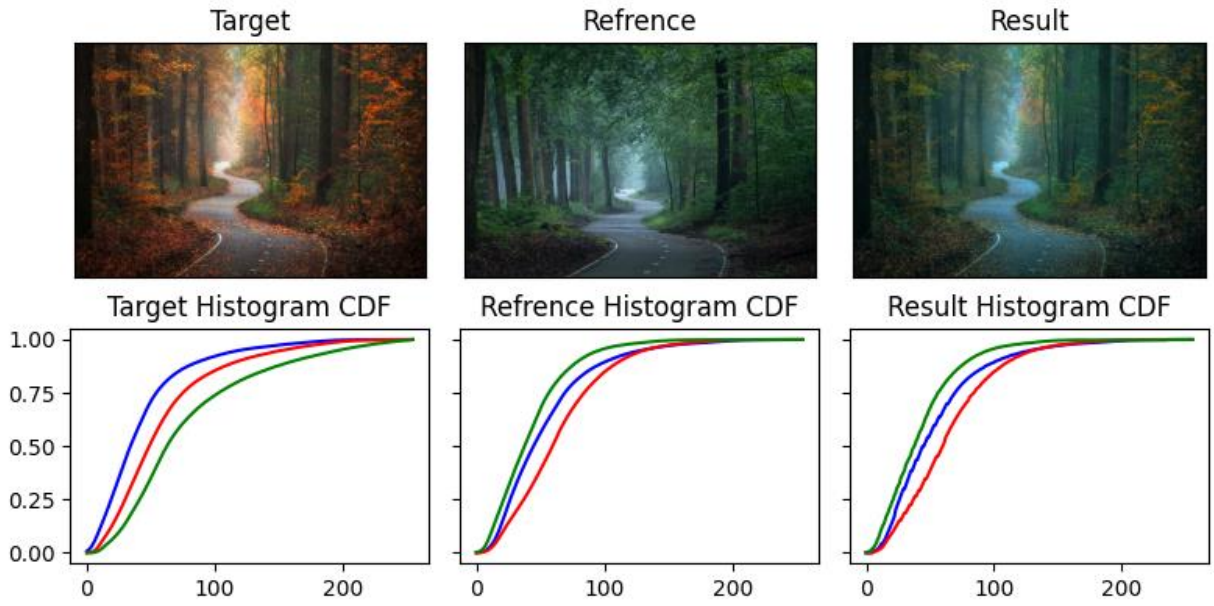
Homework 1

Due: 17/05/2023, 23:55

Please read this entire document before you begin to code or answer.

Assignment goal

In this assignment, we will implement a GPU kernel for color histogram matching (aka histogram specification). This method is used for multiple purposes, such as stitching of satellite imagery[1], medical applications, and image styling.



Given two images, “target” and “reference”, histogram matching allows changing the target image colors to look like the reference image colors. The resulting color histogram should look like the color histogram of the reference image.

For example, you can see the above three images with the CDF of their color histograms. On the left is an image of a forest during the fall (“target”). In the middle is an image of the same forest during the autumn (“reference”). On the right is the resulting image after performing histogram matching. You can see that many brown leaves became green and that the resulting histogram CDF is similar to the reference.

[1] https://medium.com/google-earth/histogram-matching-c7153c85066d

The algorithm

This is the algorithm that we will implement:

1) Create a histogram of the gray levels for each RGB color channel and for both Images (target and reference). (Total of 6 histograms.)

   Note: An RGB image is composed of three color channels: red, green, and blue. Each color channel is like a "gray image". We will use "gray levels" to refer to the pixels' values within a color channel[2].

2) Use the histograms to create three maps, one for each color channel.
3) Use these three maps to find the new color for each pixel in the target image.

See more details below.

We make the following simplifying assumptions:

- the images are square, with size $N \times N$
  - $N \geq 64$
  - $N = 2^k, k \in \mathbb{N}$
- Each image is represented as an unsigned char array of length $N \cdot N \cdot 3$, with values in the range [0, 255], where 0 means black, and 255 means white.
  For example, a color image of size 2x2 will be laid out in the array as follows:

Image:

Pixel 0: b_0, g_0, r_0	Pixel 1: b_1, g_1, r_1
Pixel 2: b_2, g_2, r_2	Pixel 3: b_3, g_3, r_3

Array:
$[b_0, g_0, r_0, b_1, g_1, r_1, b_2, g_2, r_2, b_3, g_3, r_3]$

[2] https://en.wikipedia.org/wiki/Grayscale#Grayscale_as_single_channels_of_multichannel_color_images

We calculate the map <u>for each channel</u> as follows:

1) For each image (target and reference), create a histogram h: an array of length 256. $h[v]$ is the number of pixels that have the gray level value v.
2) Calculate the prefix sum of the histogram:

$$CDF[v] = h[0] + h[1] + \cdots + h[v]$$

   Note: it's not actually the CDF because we don't normalize.

3) Calculate a map m from old to new gray levels ($m[v]$ will be the new gray level for pixels that originally had the gray level v).
   m is calculated as follows:

$$m[v] = \text{argmin}_k \left| CDF_{\text{target}}[v] - CDF_{\text{reference}}[k] \right|$$

   This step can be divided into two sub-steps:

   i. Create a matrix δ with size 256×256, such that:

$$\delta[i][j] = \left| CDF_{target}[i] - CDF_{\text{reference}}[j] \right|$$

   ii. For each gray level v find the mapping $m[v]$ by performing argmin on the row $\delta[v]$:

$$m[v] = \text{argmin}_k \delta[v][k]$$

   Note:

   - The function `argmin()` is given to you; you don't need to implement it. You can see that the implementation is similar to `reduce()` that we saw in the tutorial.
   - Since that shared memory is a limited resource, we can't store the entire matrix δ. Instead, store only part of the matrix's rows and iterate. Use just enough memory to maximize the parallelism.

Why does it work?

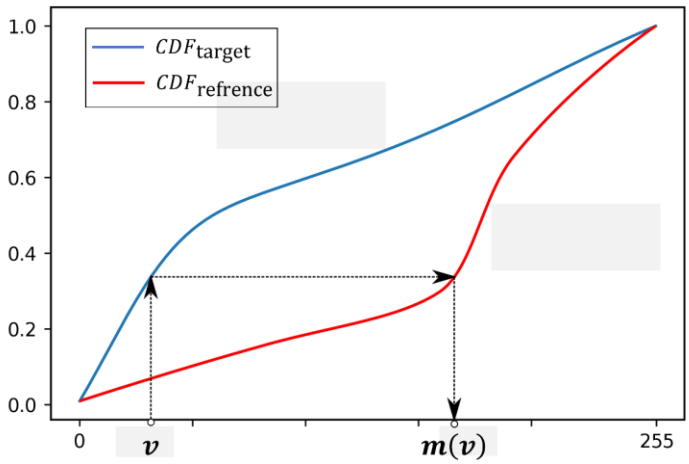
How did we come up with the following equation?

$$m[v] = argmin_k \left| CDF_{target}[v] - CDF_{reference}[k] \right|$$

Suppose that the CDF functions were continuous. Then we would like to calculate

$$m(v) = CDF_{\text{reference}}{}^{-1}\left(CDF_{\text{target}}(v)\right)$$

Which can be graphically represented as follows:



Note that for the continuous case, the following holds:

$$\text{argmin}_x \left| CDF_{\text{target}}(v) - CDF_{\text{reference}}(x) \right| = CDF_{\text{reference}}{}^{-1}\left(CDF_{\text{target}}(v)\right)$$

That is because the difference becomes zero when $x = m(v)$.

However, the CDF functions are discreet, and the inverse function does not exist, so we use the given equation as an approximation.

Homework package

Filename	Description
homework1.pdf	This file.
ex1.cu	A template for you to implement the exercise and submit. This is one of the files you are allowed to edit and the only code file you submit.
ex1.h	A header file with defines, etc. You may **not** edit this file, except for the function declarations under "#ifdef UNIT_TEST".
ex1-cpu.cu	A CPU implementation of the algorithm that is used to check the results.
main.cu	A test that runs your implementation on random images, compares the result against the CPU implementation above, and measures performance. Use it to ensure your implementation is correct and measure performance, by running "./ex1".
tests.cu	Unit tests to ease the coding process. You may edit this file. You should not submit it.
randomize_images.h, randomize_images.cu	samples random images from a non-uniform distribution. Used by main.cu and tests.cu.
image.cu	A test program that runs the CPU implementation with two given images. Produces an output image file (images/result.bmp). Use it if you'd like, by running "./image <target image> <reference image>".
Images/*.bmp	The images of the forest shown above.
Makefile	Allows building the exercise "ex1", a "tests" executable, and the graphical test application "image" using "make ex1", "make tests" and "make image" respectively. Save changes before building. Includes three switches: DEBUG, TESTS, PROFILE. Set them as needed. For example, set DEBUG=1 to allow debugging and DEBUG=0 before performance measurement. Make sure to run "make clean" after changing the switches.
Files under ".vscode " directory	Configures Visual Studio Code, allowing the IDE to be used for build (Ctrl+Shift+B) and debug (Ctrl+Shift+D).

Submission Instructions

- Submission via Moodle only.
- Make sure to submit a zip/tar.gz archive.
  The archive name should be the IDs of the students, separated with an underscore (e.g. 123456789_987654321.tar.gz)
- You will be submitting an archive with two files:
  1) ex1.cu:
     a) Contains your implementation.
     b) Make sure to check for errors, especially in 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 is 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 submit other code 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 if 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 --list-gpus`).
   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`. This should show information about your GPU.
   d) In the report: Examine the output and report the number of SMs (Multiprocessors) in the report.

2) Implement device functions:

   Notes:

   - You can use the provided unit tests in tests.cu to validate the correctness of the following device functions. The unit tests are provided for your convenience; you will not be graded according to them.
   - Feel free to change any of the following:
     - The device functions in ex1.cu, including their signature. The signatures which are listed below are optional.
     - The wrapper functions at the end of ex1.cu, including their signature.
     - The function declarations at the end of ex1.h under "#ifdef UNIT_TEST".
     - The tests in tests.cu. You can also add/remove tests.

   a) Implement the device function `prefixSum()` to calculate in-place the prefix sum of the given array:

```
__device__ int prefixSum(int *arr, int len, int tid, int threads)
```

- The arguments '`tid`' and 'threads' are necessary for calling this function concurrently by multiple "thread groups" and with multiple arrays.
- Assume that '`arr`' is stored in shared memory.

b) Implement the device function `colorHist()` that takes a color image and calculates three histograms, one for each color channel:

```
__device__
void colorHist(uchar img[][CHANNELS], int histograms[][LEVELS])
```

- use `atomicAdd_blcok()` to compute the histogram. Refer to "http://docs.nvidia.com/cuda" to learn how to use it.
- Assume that '`img`' is stored in global memory, and '`histograms`' in shared memory.

c) In the report, explain why `atomicAdd_block()` is required.

d) Implement the device function `performMapping()` that takes as input the target image and the three maps and produces the result image:

```
__device__ void performMapping(uchar maps[][LEVELS], uchar
targetImg[][CHANNELS], uchar resultImg[][CHANNELS])
```

- Assume that '`maps`' is stored in shared memory, and the images are stored in global memory.

e) Familiarize yourself with the implemented device function `argmin()` that takes as input an array of size '`LEVELS`' and finds the minimum value. The computation is performed in-place. It writes the minimum value at entry #0, and its index at entry #1. If there are duplicates, the index of the first element is chosen.

__device__ void argmin(int arr[], int len, int tid, int threads)

- the arguments '`tid`' and 'threads' are necessary for calling this function concurrently by multiple "thread groups" and with multiple arrays.

3) Implement a task serial version:

   a) Implement a kernel that takes an two images (target and reference) as two arrays from global memory and returns the processed image to another location in global memory.

      ```
      __global__ void process_image_kernel(uchar *targets,
      uchar *references,  uchar *results)
      ```

      - the number of threads should be a power of 2.
      - You will Invoke this kernel in a for loop, on a single input image at a time, with a single thread block.
      - The names of the arguments are in plural as preparation for section 4.a below.

   b) In the report, explain why your global memory accesses are coalesced regardless of the number of threads.

   c) Define the state needed for the task serial in the task_serial_context struct.

   d) Allocate necessary resources (e.g. GPU memory) in the task_serial_init function, and release them in the task_serial_free function.

   e) 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.

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

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

   h) In the report: Use NVIDIA Nsight Systems to examine the execution diagram of your code. Attach a clear screenshot to the report showing two kernels' invocations with their memory movements.
      Install NVIDIA Nsight Systems on your computer and run the profiling remotely:

      - Download from here (for Windows): https://developer.download.nvidia.com/devtools/nsight-systems/NsightSystems-2023.2.1.122-3259852.msi

- Configure a connection to the server, Specify the path of the ‘ex1’ executable, and mark the “Collect CUDA trace” checkbox.
- After you get the profile report, you will need to zoom in.

i) 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 feed the GPU with all the tasks at once.
   a) Implement a version of the kernel that supports being invoked with two arrays of images and multiple threablocks, where the block index determines which image pair each threadblock will handle. Alternatively, modify the kernel from (3) to support multiple threadblocks.
   b) Define the state needed for the bulk synchronous version in the gpu_bulk_context struct.
   c) Allocate necessary resources (e.g. GPU memory) in the gpu_bulk_init function and release them in the gpu_bulk_free function.
   d) 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.
   e) In the report: Report the execution time and speedup compared to (3.g).
   f) In the report: Attach a clear screenshot of the execution diagram from NVIDIA Nsight Systems.
   g) 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?