Spring 2020 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Image Contrast Enhancement by Adaptive Histogram Equalization

Chun-Min Chang, Shang-Yen Yeh

May 6, 2020

Abstract

This project is trying to implement a Contrast Limited Adaptive histogram equalization (CLAHE) algorithm and make it run as fast as possible. In order to improve contrast in images, make the detail of elements stand out in an acceptable time period. We are utilizing the techniques we learned from this class, including CUDA programming, unified and shared memory, atomic operations, and thread synchronization. To profile our program, we use `nvprof` to get the run time details of all GPU activities and find out which part we can make it faster. Based on the profiling analysis, we improve our implementation by the usage of shared memory, unified memory, and the proper grid-block-thread CUDA configuration. Last, we compare the serial and parallel implementation of CLAHE and found that the parallel implementation in CUDA takes 4.728 ms on 512x512 RGB images, which is 97x faster compared to the serial implementation 460.053 ms.

Link to Final Project `git` repo: https://github.com/sandersyen/ME759-Final-Project

Contents

1. General information

In this important section, please provide only the following information, in bulleted form (four bullets) and in this order:

1. home department: ECE for Chun-Min Chang and CS for Shang-Yen Yeh
2. Current status: MS student
3. Individuals working on the Final Project (include yourself)
    - Chun-Min Chang
    - Shang-Yen Yeh
4. Choose one of the following three statements (there should be only one statement here):
    - I release the ME759 Final Project code as open source and under a BSD3 license for unfettered use of it by any interested party.

2. Problem statement

Social media nowadays is having a huge part in people's daily life. People like to post pictures of their travels, their dinings, and their important moments. As a student who loves high-performance computing, it could be a nightmare to take a satisfactory picture for your girlfriend (if you have, we don't). Since we are not professional photographers, a program that can automatically beautify photos to some extent will be extremely helpful. For instance, by equalizing the histogram of the picture, CLAHE can fix our picture with some regions that are too bright or too dark.

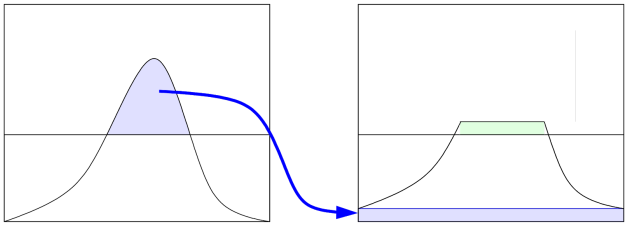
Contrast Limited Adaptive histogram equalization (CLAHE) is a common image processing technique used to adjust local contrast in a distinct section of the image by re-distributing the intensity range of pixels in that section. Unlike ordinary histogram equalization techniques, CLAHE can handle regions that are significantly brighter or darker than most regions in an image. However, it is notorious for its high computational complexity, it requires huge and repeated computations, particularly for high-resolution images. Thankfully, we can take advantage of the parallel computing skill we learned from ME759, using CUDA (GPU) to speed up the whole process and to keep the happiness in our life. We would like to implement the CLAHE algorithm in parallel by using the CUDA programming and several libraries we have learned in ME759 and compare them with the serial counterpart.

3. Solution description

We implemented the CLAHE algorithm in both Serial Implementation and CUDA Implementation. However, the overall pattern of the algorithm is the same for these two implementations. The program workflow follows the steps below:

1. Read the input image with the STB_IMAGE library. One thing needs to pay attention to is that the color space of the input image needs to be in RGB format. Our implementation is based on this premise. If the program takes an image that is not in this color space, the output will end up with wrongly cropping.

2. After we take an RGB image we will first transform the color space from RGB to LAB, since the CLAHE algorithm is applied to the lightness of the image. The method we use to do the transformation is from reference [2].
3. Split the whole image based on the input grid size.
4. For each grid, calculate the lightness histogram of the region.
5. Clip the histogram at the level of the input threshold. then redistribute the part of the histogram that exceeds the clip limit equally among all histogram bins. The redistribution will push some bins over the clip limit again, resulting in an effective clip limit that is larger than the prescribed limit. In our implementation, we leave this behavior without recursively repeating this step.



6. For each pixel find the four closest neighboring grids that surround that pixel. For the pixels lie in the rightmost and the bottom grids we pick only two closest neighboring grids. For the pixels in the bottom right corner, we pick only one closest neighboring grid.
7. Using the intensity value of the pixel as an index, find its mapping at the neighboring grids based on the CDFs of each grid.
8. Interpolate among these values to get the mapping at the current pixel location. It could be bilinear interpolation or linear interpolation depends on where the pixel locates.
9. Once we map all the lightness value for every pixel, we will transform the LAB color space back to RGB with the new lightness value and the a, b value we got from step 2.
10. Again, we use the STB_IMAGE library to write the image after CLAHE into a new file with the output name states in the argument.

Here is the summary of our implementation:

|-- main.cu	# main script to run CLAHE
|-- clahe.cuh	# header file for CLAHE algorithm
|-- clahe.cu	# parallel implementation in CUDA
|-- clahe_serial.cpp	# serial implementation in C++
|-- stb_image.h	# load images
|-- stb_image_write.h	# write images

4. Overview of results. Demonstration of your project

In both Serial Implementation and CUDA Implementation, we had run two sizes of the image to analyze the performance of each implementation. The size of the image is 512 * 512 and 1024 * 1024 respectively. The following shows the image processed by our program:

The image is processed with grid size 32 * 32 and threshold 10	
Before:	After:
	
	

A. Serial implementation:

For serial implementation, we have a basic single thread implementation for CLAHE. This is a reference to show the difference in performance between serial and parallel computations.

	512 x 512 RGB image		1024 x 1024 RGB image	
Implementation	Serial	Parallel	Serial	Parallel
transformRgbToLab clahe transformLabToRgb	102.273 ms 214.373 ms 143.407 ms	1.1168ms 20.320us 1.1415ms	340.709 ms 944.948 ms 521.65 ms	4.6534ms 74.689us 4.5064ms

B. CUDA implementation:

For CUDA implementation, we have several versions. After we finish the implementation of one version, we will run it with the sample images listed above and profile the program using nvprof to get details of each GPU activity. By having the run time for each GPU call, we try to optimize the code to make the program run faster.

a. Version 1: This is the first version of our CUDA implementation. In this implementation, we just simply let a block take care of one grid. For each block, it will calculate the histogram and CDF for the corresponding grid. For each thread, it will do the calculation for one pixel, including color space transformation and the lightness mapping. Note that in this version, we had already used the atomic operation to prevent false sharing. **The runtime (without read and write image) for 512 * 512 image is 5.09782 ms and the profiling as below:**

```
Time(%)      Time  Name
36.08%  1.1415ms  transformLabToRgb(unsigned char*, int, int, float*, float*, float*)
35.30%  1.1168ms  transformRgbToLab(unsigned char*, int, int, float*, float*, float*)
14.22%  449.89us  [CUDA memcpy HtoD]
11.44%  361.80us  [CUDA memset]
1.96%  61.984us  [CUDA memcpy DtoH]
0.64%  20.320us  clahe(float*, int, int, int, float*)
0.36%  11.297us  pixelInterpolate(float*, int, int, float*)
```

b. Version 2: From the profiling above, we learned that there are roughly two parts of the GPU activities we can work on, the CUDA memory behaviors and the kernel functions we wrote. We decided to bring shared memory into play since there are some features that a block (a grid) will share together in our kernel functions. **The runtime (without read and write image) for 512 * 512 image is 5.02778 ms and the profiling as below:**

```
Time(%)      Time  Name
36.32%  1.1406ms  transformLabToRgb(unsigned char*, int, int, float*, float*, float*)
35.53%  1.1157ms  transformRgbToLab(unsigned char*, int, int, float*, float*, float*)
14.63%  459.49us  [CUDA memcpy HtoD]
10.52%  330.50us  [CUDA memset]
1.97%  61.953us  [CUDA memcpy DtoH]
0.64%  20.160us  clahe(float*, int, int, int, float*)
```

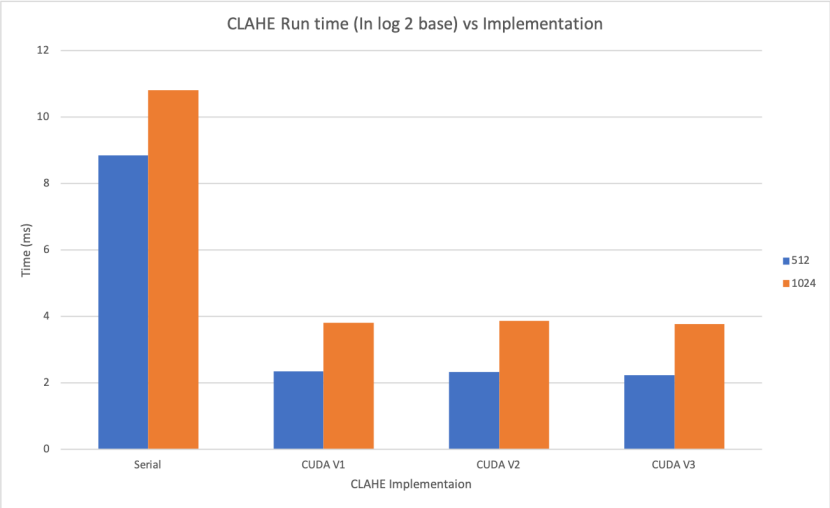
0.39% 12.192us pixelInterpolate(float*, int, int, float*)

c. Version 2: We can see the total runtime doesn't improve much. It might due to shared memory helps only kernel functions "clahe" and "pixelInterpolate". However, the majority of the time we spend is on transformation and memory behaviors. For now, it seems that there is no really good method to reduce the time the transformations need, so we try to fix the other part, memory behaviors. First, we think it might be safer to initialize all the array we have. Therefore we decided to keep [CUDA memset]. Second, there is no way to know how large the image is until we read the memory so that we couldn't allocate CUDA memory before we use a host pointer to receive the value of the image. Thus, we can not remove [CUDA memcpy HtoD]. The only behavior we can remove here is [CUDA memcpy DtoH]. By using the unified memory, we don't need to copy the value from device back to the host. We can just use the unified memory to store our new RGB value and write it into a new image file. **Now we get the runtime (without read and write image) for 512 * 512 image as 4.72829 ms and the profiling as below:**

Time(%) Time Name
35.97% 1.1417ms transformLabToRgb(unsigned char*, int, int, float*, float*, float*)
35.17% 1.1165ms transformRgbToLab(unsigned char*, int, int, float*, float*, float*)
17.37% 551.40us [CUDA memcpy HtoD]
10.46% 332.16us [CUDA memset]
0.64% 20.321us clahe(float*, int, int, int, float*)
0.39% 12.256us pixelInterpolate(float*, int, int, float*)

C. Summary:

In sum, CUDA implementations are way better than Serial Implementation. We achieve about 97x faster for 512 * 512 image. The new CUDA implementation version is always faster than our previous one. However, the improvement is too huge from Serial to CUDA, which make the progress between CUDA version be not that obvious.



5. Deliverables

- The `git` repo: https://github.com/sandersyen/ME759-Final-Project/tree/master
- The git repo will contain several subfolders, they will be organized as follows:

Folder name	Description
Serial implementation	serial implemented CLAHE with C++
CUDA implementation	parallel implemented CLAHE with CUDA programming
Test images	test images we use for this report, containing a 512 * 512 image and a 1024 * 1024 image

- For folder "Serial implementation" and "CUDA implementation", there is a run.sh file. To compile and run the programs, just type "sbatch run.sh" in Euler, it already includes the compile command and the command to run the CLAHE program.
- **NOTE: User should specify the input arguments in the script file with this pattern ./program_name [input_img_name] [output_img_name] [grid_size] [threshold]**

6. Conclusions and Future Work

In this project, we implement a known image processing algorithm, CLAHE, in both serial and parallel ways. There are several thread synchronization issues in the parallel implementation, such as in the computation of histogram and cumulative density function. While carrying out our implementation, we carefully handle these synchronization issues and also take spatial locality and temporal locality into consideration, especially when accessing large image arrays (row-majored 1-D arrays). To improve parallel implementation in CUDA, we use shared memory and unified memory to reduce potential overheads of data migration. Our parallel result is surprisingly good and outperforms the serial one by a significant margin. In our future plan, we will experiment with the CUB library to further speed up our parallel implementation. Also, we plan to accelerate the serial implementation on CPU by using multiple threads with OpenMP.

References

[1] CUDA best practice guide: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
[2] CLAHE algorithm: https://en.wikipedia.org/wiki/Adaptive_histogram_equalization
[3] Conversion between RGB and LAB: http://www.easyrgb.com/en/math.php
[4] Serial implementation reference: http://www.cs.utah.edu/~sshankar/cs6640/project2/clahe