GPGPU programming with CUDA - TP Project: Histogram equalization

Viet-Huy HA
January 7, 2023

1 Introduction

In this project, we proceed to adjust the contrast of the image by applying given algorithms and techniques that work with both CPU and GPU. We will solve the problem with 5 main steps:

1. Convert from pixels to HSV (rgb2hsv)
2. Perform histogram according to the value of V
3. From the histogram, proceed to repart, applies the repartition function r(l)
4. From repart, "spread" the histogram
5. Convert from HSV to pixels (V has been recalculated)

2 Implementation and Analysis

In this section, there will be two versions of the program mentioned: **Sequential and Parallel**. First, we will analyze in detail how to implement these two versions. Clearly state the function of the code, the program's flow. Next, We'll take a look at their efficiency, processing power, optimisation, and runtime. Then compare to be able to see the advantages and disadvantages of each version.

2.1 Sequential version - CPU version

In the sequential version, we will use loops to calculate the result for each function

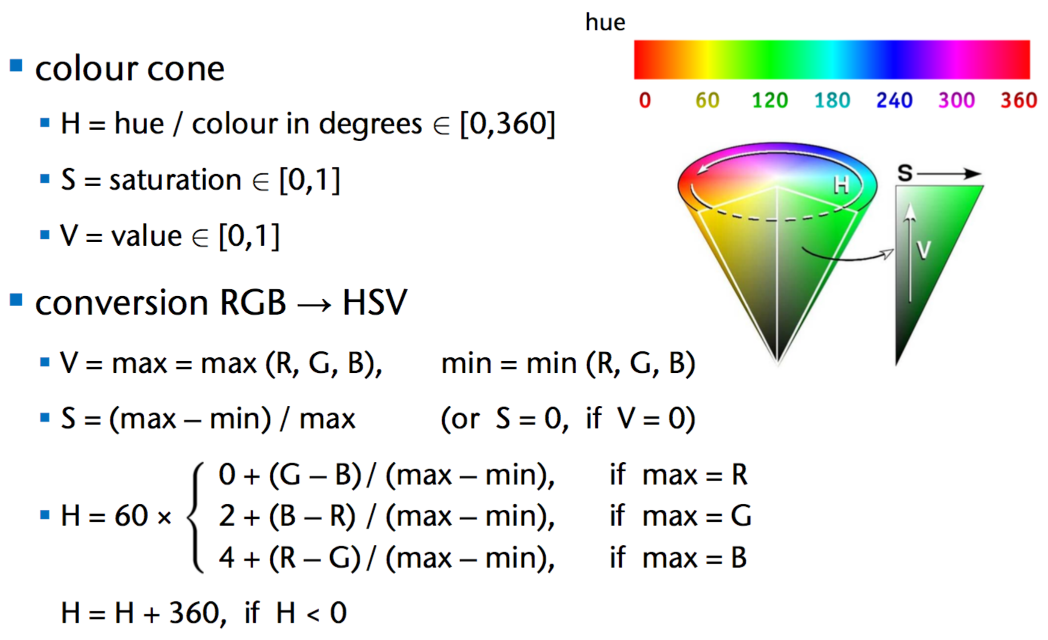
1. Convert from pixels to HSV - $rgb2hsv\_CPU$: Based on the conversion algorithm from RGB to HSV here: https://en.wikipedia.org/wiki/HSL_and_HSV. We have successfully converted from RGB to three arrays h, s, v.
2. Perform histogram according to the value of V - $histogram\_CPU$: By using array of markers, we successfully generate histogram according to the value of V.
3. From the histogram, proceed to repart, applies the repartition function r(l) - $repart\_CPU$:

$$r(l) = \Sigma_{k=0}^{l} h(k)$$

4. From repart, "spread" the histogram - $equalization\_CPU$:

$$T(x_i) = \frac{L-1}{L \times n} \times r(x_i)$$

5. Finally, we convert hsv to RGB (V has been recalculated) - $hsv2rgb\_CPU$: We use following algorithm:



- colour cone
  - H = hue / colour in degrees ∈ [0,360]
  - S = saturation ∈ [0,1]
  - V = value ∈ [0,1]
- conversion RGB → HSV
  - V = max = max (R, G, B),   min = min (R, G, B)
  - S = (max – min) / max   (or S = 0, if V = 0)
  - $$H = 60 \times \begin{cases} 0 + (G - B) / (\max - \min), & \text{if } \max = R \\ 2 + (B - R) / (\max - \min), & \text{if } \max = G \\ 4 + (R - G) / (\max - \min), & \text{if } \max = B \end{cases}$$

  H = H + 360, if H < 0

Figure 1: hsv2rgb Algorithm

6. Time execution: **48.0108 ms**

2.2 Parallel version - GPU version

In this section, we will use programming techniques on the GPU. Therefore, we will go into more detail on each function.

2.2.1 Kernel $rgb2hsv$

About the algorithm idea, this part is no different from the CPU version. What we need to pay attention to in this section is to install it on the $kernel$ and parallel programing. Therefore, I used version as follows:

- Arguments:
  1. $const\, unsigned\, char * dev\_pixels\_in$: pixel table of input image (as 3 bits R, G, B in series)
  2. $double * dev\_h, double * dev\_s, double * dev\_v$: the table containing the output of the H, S, and V values respectively after being calculated
  3. $const\, int\, imgSize$: size of input image
- We initialized $kernel$ with 512 threads per block and number of blocks was calculated by

$$\frac{imgSize + numberThread - 1}{numberThread}$$

- Calculate current index of input pixels array $tid = blockIdx.x * blockDim.x + threadIdx.x$
- Then, perform the algorithm on each thread for each pixel by accessing array dev_pixels_in (global memory) and return to three array dev_h, dev_s, dev_v (global memory)

2.2.2 Kernel $hsv2rgb$

Similar to the initialization of rgb2hsv and the algorithm is the same as that used in CPU

- Arguments:

1. $double * dev\_h, double * dev\_s$: the table containing the output of the H and S from function $rgb2hsv$
2. $unsigned\,char * dev\_pixels\_out$: pixel table of output image (as 3 bits R, G, B in series)
3. $const\,int\,imgSize$: size of input image

- We initialized $kernel$ with 512 threads per block and number of blocks was calculated by

$$\frac{imgSize + numberThread - 1}{numberThread}$$

- Calculate current index of input pixels array $tid = blockIdx.x * blockDim.x + threadIdx.x$
- Then, perform the algorithm with each pixel

2.2.3 Kernel $histogram$

The idea of the algorithm used in this section is the same as the CPU version. But we will pay attention to the optimization with each version on the GPU (optimization, timing, ...). Here are a two versions I recommend:

Simple Version - $kernelCompute\_histogram$

- This version only applies parallel computing in the most basic way
- We initialized $kernel$ with 512 threads per block and number of blocks was calculated by

$$\frac{imgSize + numberThread - 1}{numberThread}$$

- Each thread will be responsible for checking for each position in the input array
- To avoid conflict, we use $\_\_syncthreads()$ before incrementing the marker array by 1
- Time execution **2.40886 ms**

Optimization Version - $kernelCompute\_histogram\_v2$

- In this version we will use it in combination with $sharedmemory$
- We initialized $kernel$ with 512 threads per block and number of blocks was calculated by

$$\frac{imgSize + numberThread - 1}{numberThread}$$

- Initialize a $sharedmemory$ array to keep track for each index of input array
- We will make use of initialization loop $threadIdx.x$, boundary is the largest possible index of tracking array, hop is $blockDim.x$, i.e. 512. Because at most there are only 256 elements in tracking array $blockDim.x = 512$, so this loop only resets each value at an index once. When the program progress to another block, shared memory array will be created new and reset
- First $\_\_syncthreads()$ was implemented to avoid conflicts when using shared memory.
- Second $\_\_syncthreads()$ was implemented to avoid conflicts when saving data to output array
- Time execution: **2.26835 ms**
- Obviously, this version is more efficient than the simple version and saves execution time

2.2.4 Kernet $repart$

From the histogram, proceed to repart, applies the repartition function r(l) - $kernelCompute\_repart$:

$$r(l) = \Sigma_{k=0}^{l} h(k)$$

In this part, we will create two versions in order to compare efficient:

Simple Version - $kernelCompute\_repart$

- We initialized $kernel$ with 256 threads and number of blocks was calculated by

$$\frac{256 + numberThread - 1}{numberThread}$$

We only need to use one block, because max index of repart array is 255, and size is 256

- Each thread will be responsible for calculating an index of the repart array by applying the above formula

Optimization Version - $kernelCompute\_repart\_v2$

- We also use algorithm like $Simple\,Version$, but combine with $shared\,memory$. We can reduce time to calculate repart array by accessing shared memory instead of global memory.

2.2.5 Kernel $equalization$

After ”repart”, We implement a equalization, from the previous distribution, “spread” the histogram by using parallel programing.

- We initialized $kernel$ with 512 threads per block and number of blocks was calculated by

$$\frac{imgSize + numberThread - 1}{numberThread}$$

- We will use the same fomula with CPU:

$$T(x_i) = \frac{L-1}{L \times n} \times r(x_i)$$

- Each thread will be responsible for calculating an index of T array

2.3 Result and Comparison between two versions

After completing the work, we get result as follows:

1. Successfully adjusted the contrast of input image:

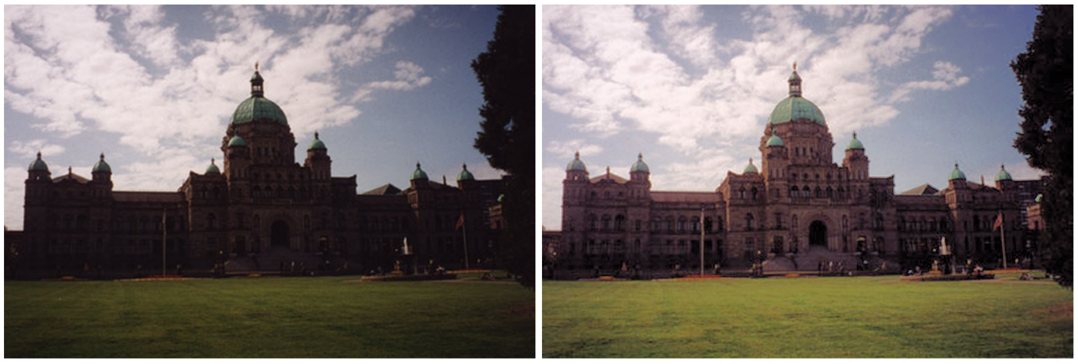


Figure 2: Chateau.PNG

2. Difference between Histogram before and after calculation:

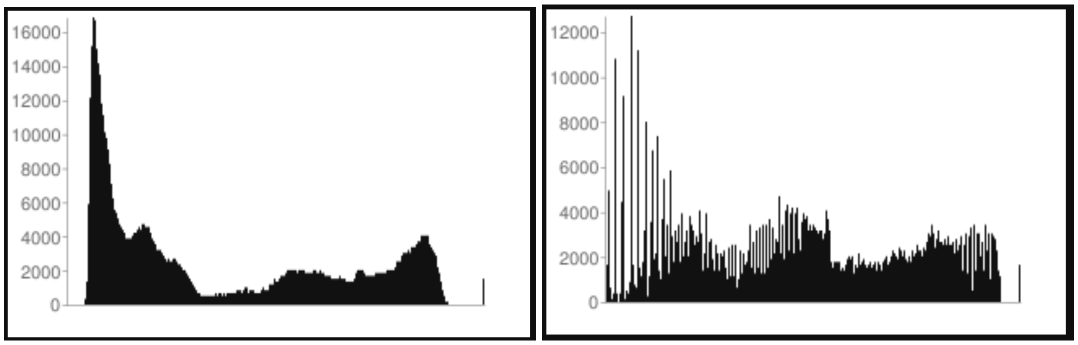


Figure 3: Histogram.PNG

3. Time execution

CPU	48.0108 ms
GPU Simple	2.40886 ms
GPU Optimization: histogram_v2	2.26835 ms
GPU Optimization: histogram_v2 + repart_v2	2.1532 ms

Regarding the comparison between the CPU and GPU versions, we can see that the GPU is far superior in terms of efficiency and computation time. Although the implementation of algorithms in parallel programming is quite complicated, it is very effective.
In addition, the use of shared memory and synchronization methods significantly increased the performance of the program on the GPU, avoiding some conflicts