Programmation GPU

Anh-Kiet DUONG

January 3, 2023

Contents

Acronyms

CDF Cumulative Distribution Function

CPU Central processing unit

GPU Graphics processing unit

HSV Hue-Saturation-Value

MAS Max Absolute Scale

MMS Min-max Scale

RGB Blue-Green-Red

RGB Red-Green-Blue

1 Introduction

1.1 General

My name: Anh-Kiet DUONG.
Program: Master (M1) CRYPTIS.
Demo:

- Lab computer Quadro P620
- GeForce MX130 Windows
- Google Colab Tesla T4 Linux
- Google Colab Tesla A100 Linux

Source code: github

1.2 Program

The source code can be compiled and run on Google Colab, personal computer (Windows, Linux), and lab machine I.211-212,...

The program take 11 parameters: Input image filename, Output image filename, Scaler (0=MaxAbsScaler, 1=MinMaxScaler), nbHistogram, hardware CPU/GPU (0=CPU, 1=GPU), GPU gridDim.x, GPU gridDim.y, GPU gridDim.z, GPU blockDim.x, GPU blockDim.y, GPU blockDim.z. For example,

```
./main images/Chateau.png images/Chateau_GPU.png 1 256 1 8 1 1 8 1 1
```

The command call program with the input image "images/Chateau.png", output file "images/Chateau_GPU.png", scaler: Min-max Scale, $nbHistogram = 256$, hardware GPU, $gridDim = (8, 1, 1)$, $blockDim = (8, 1, 1)$

```
./main images/Chateau.png images/Chateau_CPU.png 0 256 0
```

The command call program with the input image "images/Chateau.png", output file "images/Chateau_CPU.png", scaler: Max Absolute Scale, nbHistogram 256, hardware CPU.

2 CPU

2.1 rgb2hsv

Follow the steps in the method mentioned in the topic to convert RGB to HSV. [1]. For each pixel, we will do it once. So the algorithm has time complexity $O(h \times w)$, and space complexity $O(h \times w)$ (input/output arrays). Where h, w is the height and width of the input image.

2.2 hsv2rgb

Follow the steps in the method mentioned in the topic to convert HSV to RGB. [2]. For each pixel, we will do it once. So the algorithm has time complexity $O(h \times w)$, and space complexity $O(h \times w)$ (input/output arrays).

2.3 histogram

We iterate through all the pixels and add 1 to the corresponding position of the resulting array. So the algorithm has time complexity $O(h \times w)$, and space complexity $O(h \times w + nbHistogram)$ (input/output arrays). Where $nbHistogram$ is the number of histogram's bins.

2.4 repart

We iterate through all the elements of the histogram list from 1 to $nbHistogram$, each element we add it and the one before it. So the algorithm has time complexity $O(nbHistogram)$, and space complexity $O(nbHistogram)$ (input/output arrays).

We will have 2 ways to scale the value of $r(x_i)$ to the range $[0, 1]$. To see more clearly the difference between the two methods, see section 4.1.3.

2.4.1 Max absolute scale

In the simplest way MAS, for non-negative integers in the range [0, 1] we divide them by their maximum. This is also the method outlined in the request of the project.

$$x_i = \frac{x_i}{x_{max}} \tag{1}$$

where x_i is values index i^{th} and x_{max} is the maximum of the ensemble X.

[1] https://en.wikipedia.org/w/index.php?title=HSL_and_HSV&section=25
[2] https://en.wikipedia.org/w/index.php?title=HSL_and_HSV&section=18

2.4.2 Min-max scaler

In addition, we also have another way to scale which is the MMS where at least one value will be scaled to zero.

$$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}} \tag{2}$$

where x_{min} is the minimum of the ensemble X.

2.5 equalization

There will be 2 ways to implement. The simple way for each pixel we will perform division to bring the pixel back range $[0, 1]$. Or we can do the division first and assign the result and the CDF array. Because division also does not take too much complexity (because for each pixel we also have an assignment). So it reduces many calculation steps, and the speed is faster (not too much).

3 GPU

3.1 rgb2hsv

The algorithm is the same as the previous section 2.1. But this time because there are many threads, instead of a sequential loop, we will let each thread handle a number of pixels (divided equally by each thread). So the algorithm has time complexity $O\left(\left\lceil \frac{h \times w}{nThreads} \right\rceil\right)$ and space complexity $O\left(h \times w\right)$ (input/output arrays). Where $nThreads$ is the total number of threads.

Bank conflicts [3]

There are two versions. Version 0 (*rgb2hsv_v0*) is not optimized for bank conflicts. And version 1 (*rgb2hsv*) is the best one.

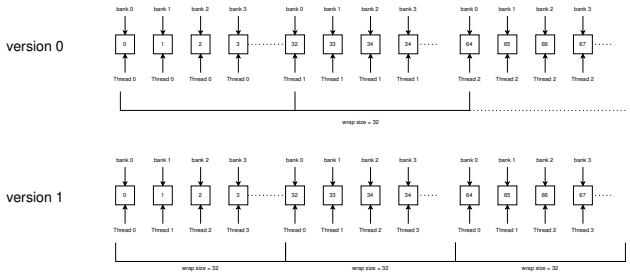
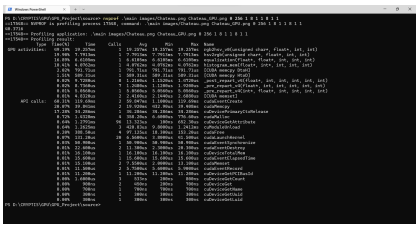


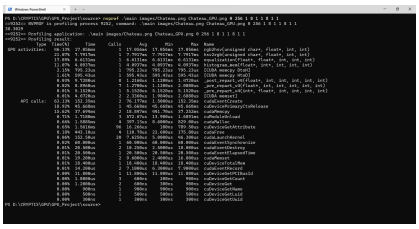
Figure 1: Bank conflicts example.

Command: *nvprof . main images/Chateau.png Chateau_GPU.png 0 256 1 8 1 1 8 1 1*

[3] https://github.com/Kobzol/hardware-effects-gpu/blob/master/bank-conflicts/README.md



(a) Version 0. Avg: 19ms



(b) Version 1. Avg: 17ms

Figure 2: Performance of version 0 and version 1. Because it's too long, I need to shrink the image. Please zoom in for an exact view.

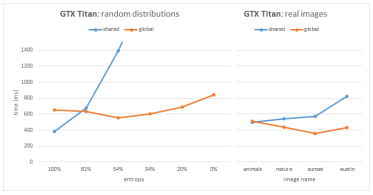
For all other functions, I just applied the best bank optimization and don't repeat the comparison.

3.2 hsv2rgb

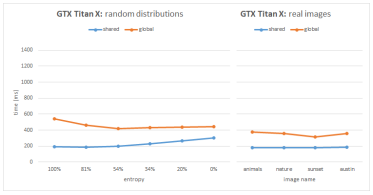
The algorithm is the same as the previous section 2.2. And the same implementation with 3.1. So the algorithm has time complexity $O\left(\left\lceil \frac{h \times w}{nThreads} \right\rceil\right)$ and space complexity $O\left(h \times w\right)$ (input/output arrays).

3.3 histogram

As we did in TP3, we can see that using shared memory makes computation much faster. Because atomic is the same principle as semaphore, where only one thread can modify the resource at a time. Therefore, creating an array at each block will reduce the chance of collisions. In an official NVIDIA article [4], they also talked about the same thing on Maxwell GPUs.



(a) Kepler architecture NVIDIA GeForce GTX TITAN GPU



(b) Maxwell architecture NVIDIA GeForce GTX TITAN X GPU

Figure 3: Performance of histogram algorithm comparing global memory atomics to shared memory atomics. The image is excerpted from NVIDIA's post.

In this case, the complexity also depends on the collision probability of the threads. Hence in the best case (no collision), the time complexity is

[4] https://developer.nvidia.com/blog/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/

$O\left(\left\lceil \frac{h \times w}{nThreads} \right\rceil\right)$. And in the worst case (all collisions), the time complexity is $O\left(\left\lceil \frac{h \times w}{bBlocks} \right\rceil + nbHistogram * nBlocks\right)$. In real, calculating the time complexity, in this case, will be much more complicated. And the space complexity is $O\left(h \times w + nbHistogram + nbHistogram * nBlocks\right)$ respectively input image, result array, block shared arrays.

3.4 repart

I have referenced a resource: github [5]. However, I also found some of their problems. For simplicity, we omit the *"bank"* parameter in the complexity calculation. Instead, I would say which is faster for the same time complexity.

- Problem1: Their Kogge-Stone implementation is definitely wrong. And when run there is a probability of incorrect results because the array may have been modified by one thread before it is called in another thread. As in the example figure 4, thread 1 transforms 2 to 3 at $a[1]$ before thread 2 calls $a[1]$. When step (2^i) is bigger than the GPU warp size, it's fine.
- Problem 2: Their implementation is a fixed number of blocks and a fixed number of threads. Suppose, $nbHistogram$ is a larger number than the number of threads in a block (to make it smoother maybe, but normally we just set it to 256). Or there is a GPU that is too small to accommodate many threads. In general, we need a more comprehensive solution.

3.4.1 Baseline

In the simplest way, for each index in the hist array, we need to iterate from 0 to itself to calculate the sum. So we assign work to each thread that will receive an index number (divided equally). Time complexity is $O\left(nbHistogram \times \left\lceil \frac{nbHistogra}{nThreads} \right\rceil\right)$, and space complexity $O\left(nbHistogram\right)$ (input/output arrays). It's even worse than the CPU dynamic programming version 2.4.

3.4.2 Optimize baseline

In addition, we can also set up from the end to the index because the sum will be equal to $h \times w$. Therefore, we divide it into two parts. One part iterates from the beginning, the other from the end, and chooses whichever is faster. Time complexity is $O\left(nbHistogram \times \left\lceil \frac{nbHistogra}{nThreads} \right\rceil\right)$, and space complexity $O\left(nbHistogram\right)$ (input/output arrays). It's even worse than the CPU dynamic programming version 2.4.

On the other hand, using more shared memory also helps to speed up computation (*repart_v2_mem*). And the space complexity increase to $O\left(nbHistogram \times nThreads\right)$. No need for memory output because we only write once. If we have memory, we still need at least 1 write.

[5] https://github.com/nuwandda/cuda-histogram-equalization

3.4.3 Brent–Kung

We use the Brent–Kung adder[1]. Take a look at my calculation example below for a better understanding.

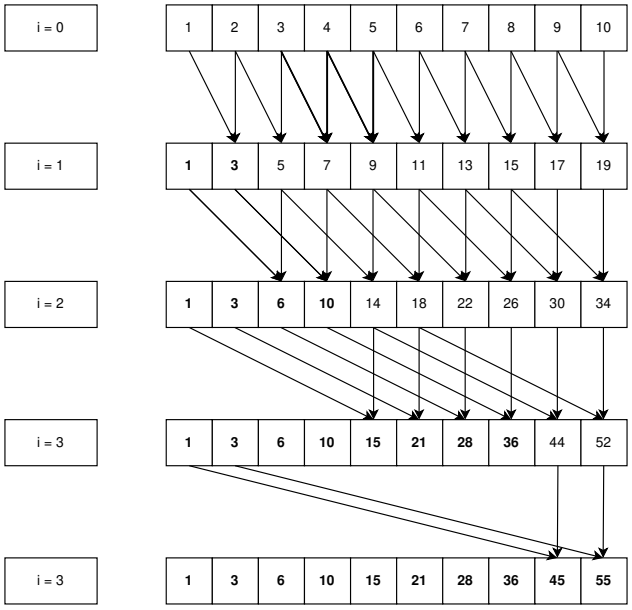


Figure 4: Brent–Kung example

To avoid the problem 3.4 as in their repo stated above. First, we need 2 arrays so that the data won't be written before it is called. Second, we need to make sure i = 0 is done. However, when other than threading blocks, we cannot use *__syncthread()*. So we need a function like *__syncBlock()* for example. However, this function doesn't exist, so I ended up wrapping a *__global__* function into *__syncBlock()*. And we will make the function call many times. Each function termination is a *__syncBlock()*.

So the time complexity is $O\left(\left\lceil \log_2\left(nbHistogram\right) \times \left\lceil \frac{nbHistogram}{nThreads} \right\rceil \right\rceil\right)$ and space complexity is $O\left(nbHistogram\right)$ (even when not count input/output). So from now on it can be faster than the CPU sequential version 2.4.

3.4.4 Kogge–Stone

We use the Kogge–Stone adder[2].

Similar to 3.4.3, but at each step we only do a half of 4 to avoid problem 1 3.4. Problem 2 3.4 is still the same and the solution is similar to 3.4.3.

So the time complexity is $O\left(\left\lceil \log_2\left(nbHistogram\right) \times \left\lceil \frac{nbHistogram}{nThreads} \right\rceil \right\rceil\right)$ and space complexity is $O\left(1\right)$ (when not counting input/output) or $O\left(nbHistogram\right)$ (when count input/output).

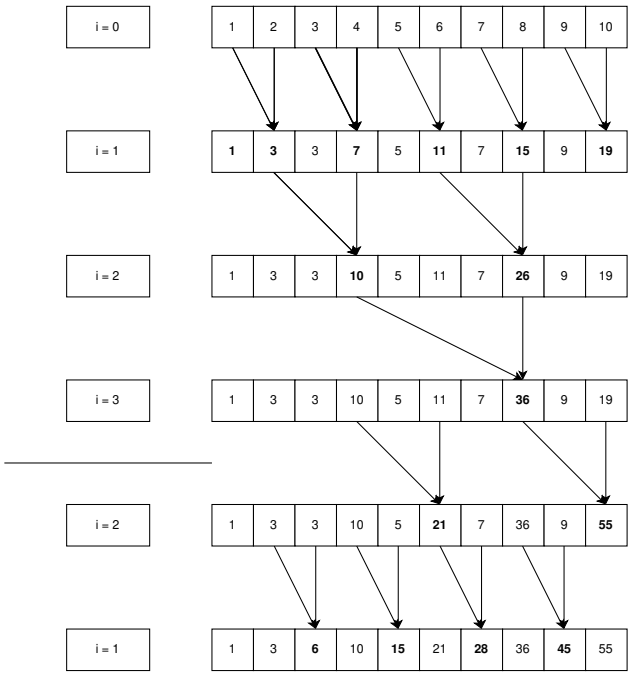


Figure 5: Kogge–Stone example

3.4.5 Kogge–Stone shared memory

In the case of $nBlocks = 1$ or $nThreads > nbHistogram$ we will see that **shared memory** is more efficient. So we will call the *repart_v4_mem* function. Now the space complexity is $O(nbHistogram)$ (even when not counting input/output) and time complexity is the same but a little bit faster because of memory access speed.

3.5 equalization

Similar to section 3.2 and 3.1 we divide equally among each thread that will process the pixels. So the algorithm has time complexity $O\left(\left\lceil \frac{h \times w}{nThreads} \right\rceil\right)$ and space complexity $O(h \times w)$ (input/output arrays).

4 Result

4.1 Histogram

To make the chart easier to see, we do MAS to reduce the total histogram (red) line maximum equal to max(hist)

4.1.1 Chateau

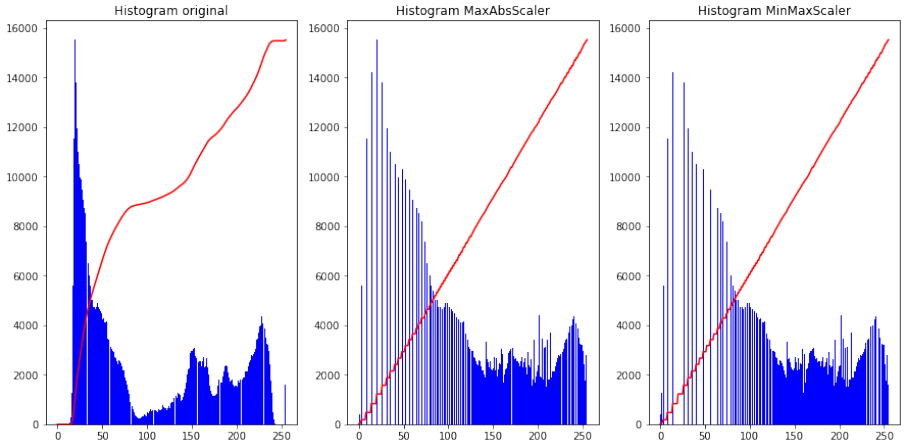


Figure 6: Chateau image and histogram. The red line is the sum of the histogram scaled.

	MaxAbsScale	MinMaxScale
CPU	7f698b2c308e2b3a4c2fb264f21360c7	7f698b2c308e2b3a4c2fb264f21360c7
GPU	7f698b2c308e2b3a4c2fb264f21360c7	7f698b2c308e2b3a4c2fb264f21360c7

Table 1: Compare MD5 hex digest [3] of file generated by CPU and GPU.

4.1.2 Palais GarnierFichier

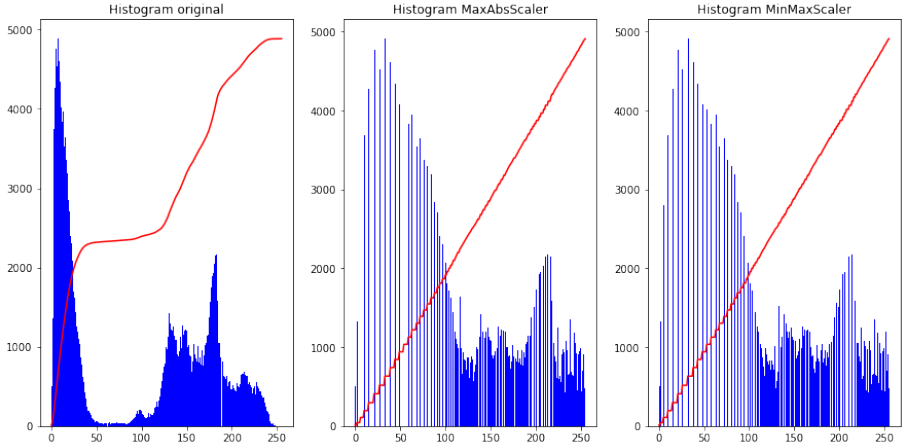


Figure 7: Palais GarnierFichier image and histogram. The red line is the sum of the histogram scaled.

	MaxAbsScale	MinMaxScale
CPU	8beee4e1cac3512c9ebb9f736b8e4a52	23baaefcdc8477db735f9990630b5abc
GPU	8beee4e1cac3512c9ebb9f736b8e4a52	23baaefcdc8477db735f9990630b5abc

Table 2: Compare MD5 hex digest [3] of file generated by CPU and GPU.

4.1.3 Lenna

We use the following code to generate an image that has been changed in brightness.

```
import cv2

def change_brightness(img, value=30):
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    v = cv2.add(v,value)
    v[v > 255] = 255
    v[v < 0] = 0
    final_hsv = cv2.merge((h, s, v))
    img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
    return img

image = cv2.cvtColor(cv2.imread("images/Lenna.png"),
↪  cv2.COLOR_BGR2RGB)
image_brightness = change_brightness(image, -120)
cv2.imwrite("images/Lenna_brightness.png",
↪  cv2.cvtColor(image_brightness, cv2.COLOR_RGB2BGR))

plt.figure()
f, axarr = plt.subplots(1,2)
axarr[0].imshow(image)
axarr[0].set_title("Origin Lenna")
axarr[0].axis('off')

axarr[1].imshow(image_brightness)
axarr[1].set_title("Lenna increase brightness")
axarr[1].axis('off')

plt.show()
```

Figure 8: Lenna image before and after change brightness.

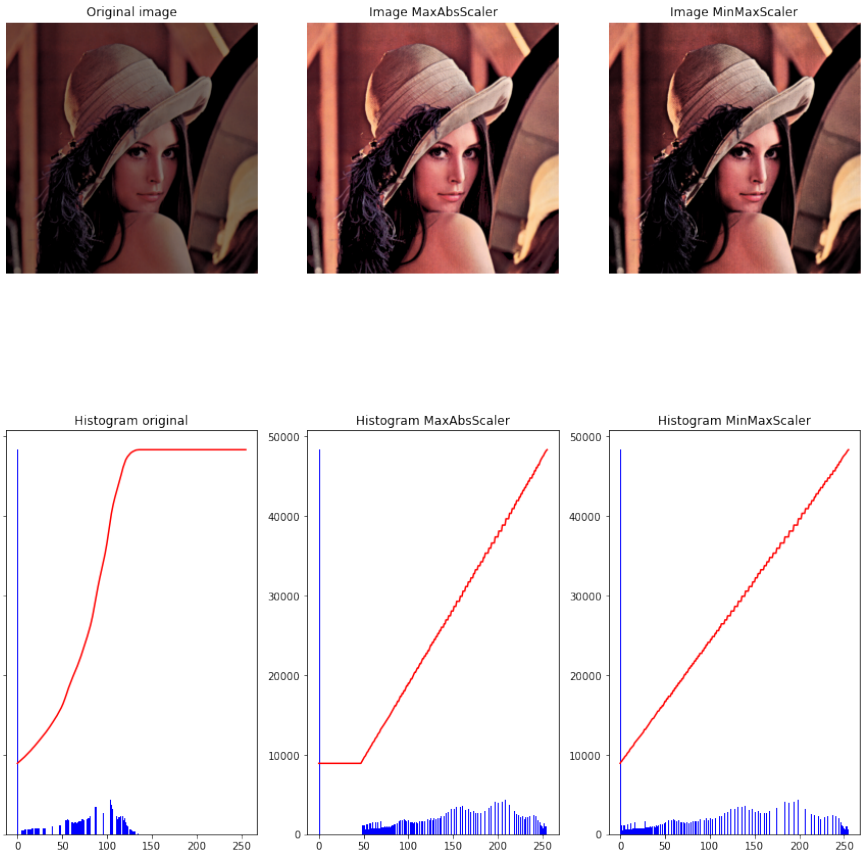


Figure 9: Lenna image and histogram. The red line is the sum of the histogram scale.d

We can see that this time the algorithms MAS and MMS have a difference. However, in terms of output image quality, there is not too much variation. You can look at her hat to see the difference.

4.2 Performance

4.2.1 CPU

Mean CPU time:

1. Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz (Lab CPU): 56.78
2. Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz (Personal CPU): 191.93
3. Intel(R) Xeon(R) CPU @ 2.00GHz (Google Colab GPU): 74.94

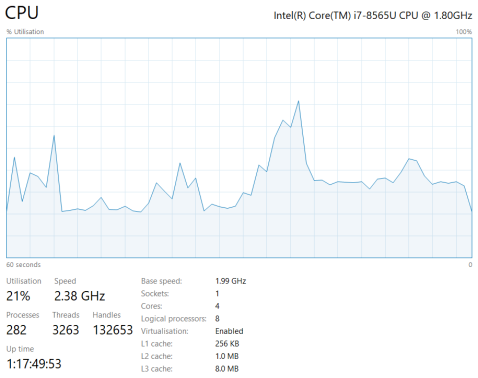


Figure 10: CPU usage after 10 times call program in mode CPU.

4.2.2 GPU

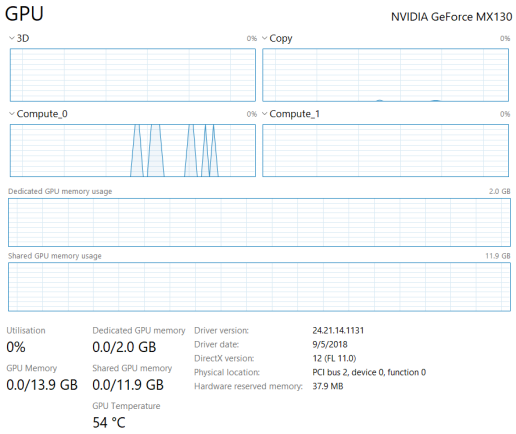


Figure 11: GPU (Nvidia GeForce MX130) usage after 10 times call program in mode GPU

5 Time

5.1 CPU vs GPU

5.1.1 Nvidia Quadro P620 vs Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz

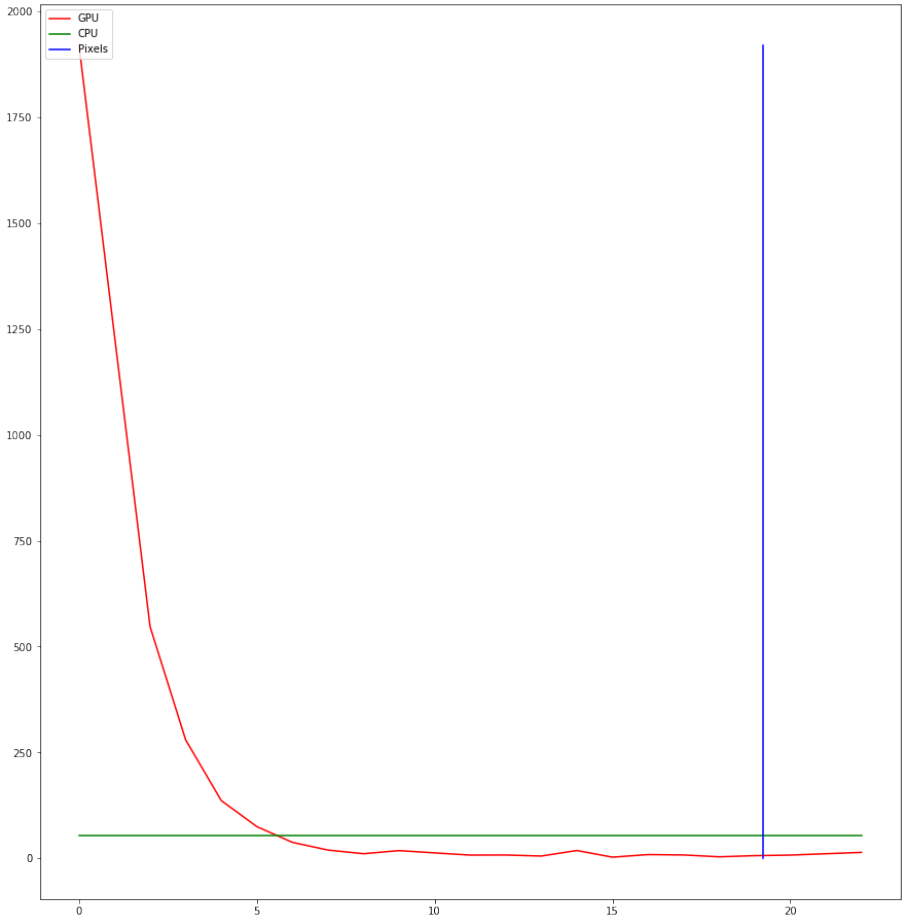


Figure 12: Time execute of Nvidia Quadro P620 vs Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz. The vertical axis is running time, the horizontal axis is $log_2\left(nThreads\right)$. The green line is CPU time. The red line is GPU time. The blue is $log_2\left(h \times w\right)$

5.1.2 Nvidia GeForce MX130 vs Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

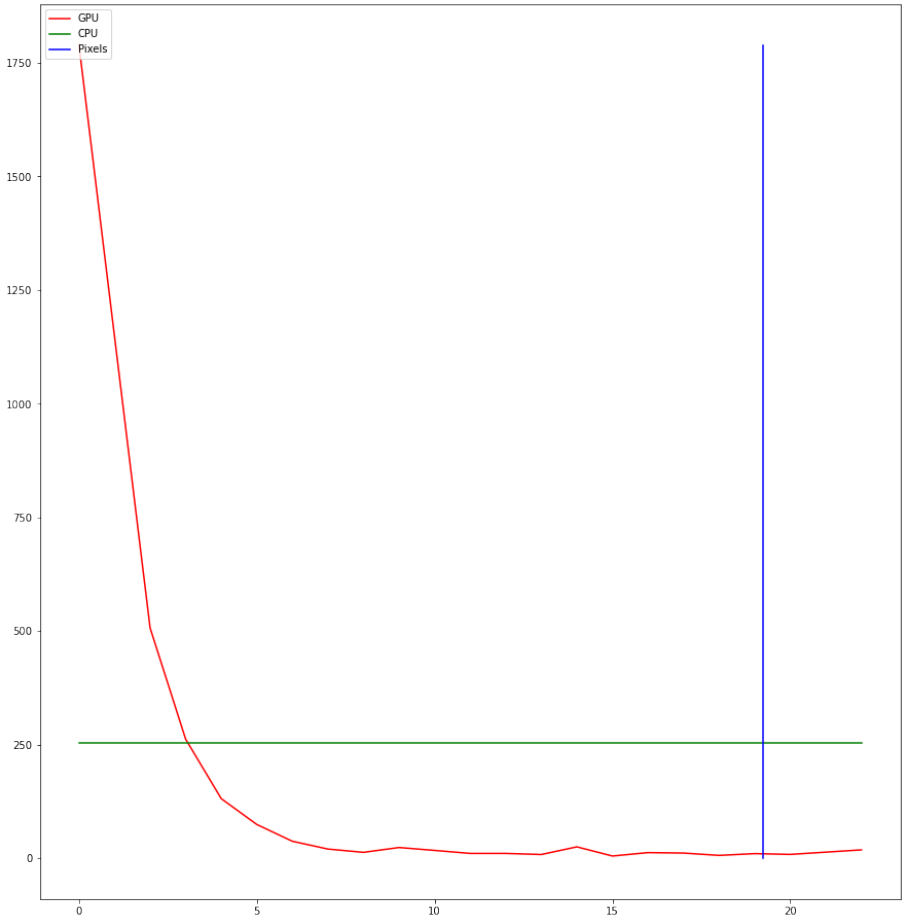


Figure 13: Time execute of Nvidia GeForce MX130 vs Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz. The vertical axis is running time, the horizontal axis is $log_2\left(nThreads\right)$. The green line is CPU time. The red line is GPU time. The blue is $log_2\left(h \times w\right)$

5.1.3 Nvidia Tesla T4 vs Intel(R) Xeon(R) CPU @ 2.00GHz

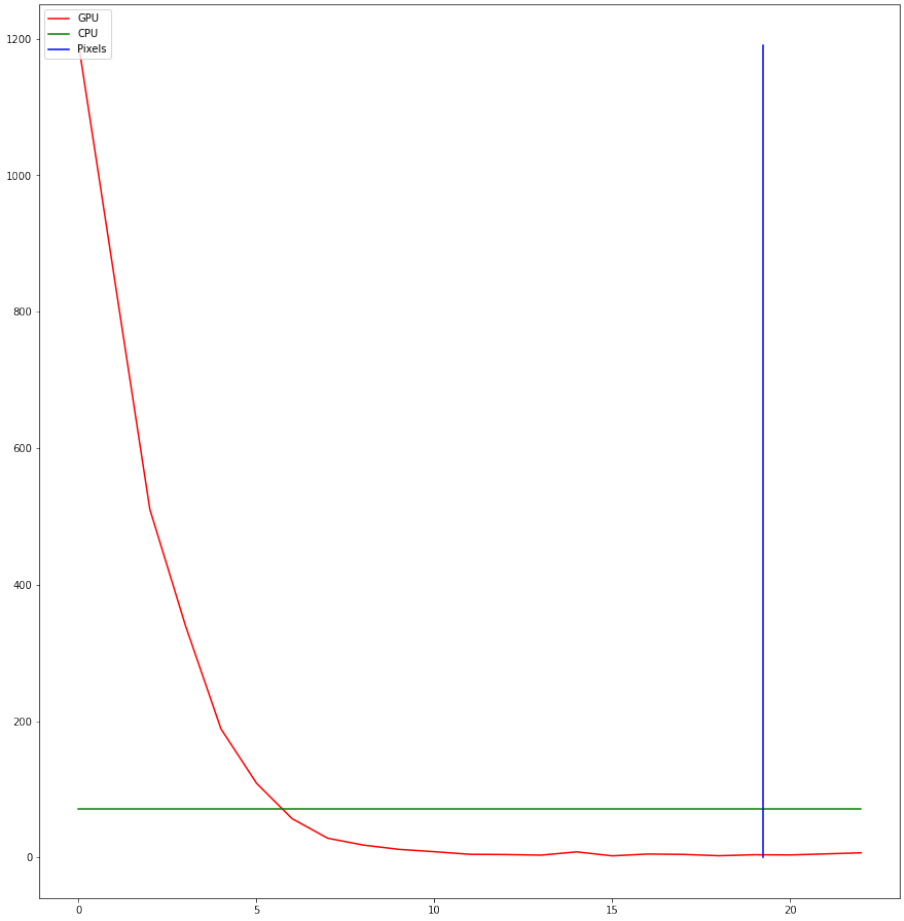


Figure 14: Time execute of Nvidia Tesla T4 vs Intel(R) Xeon(R) CPU @ 2.00GHz. The vertical axis is running time, the horizontal axis is $log_2\left(nThreads\right)$. The green line is CPU time. The red line is GPU time. The blue is $log_2\left(h \times w\right)$

5.1.4 Nvidia Tesla A100 vs Intel(R) Xeon(R) CPU @ 2.00GHz

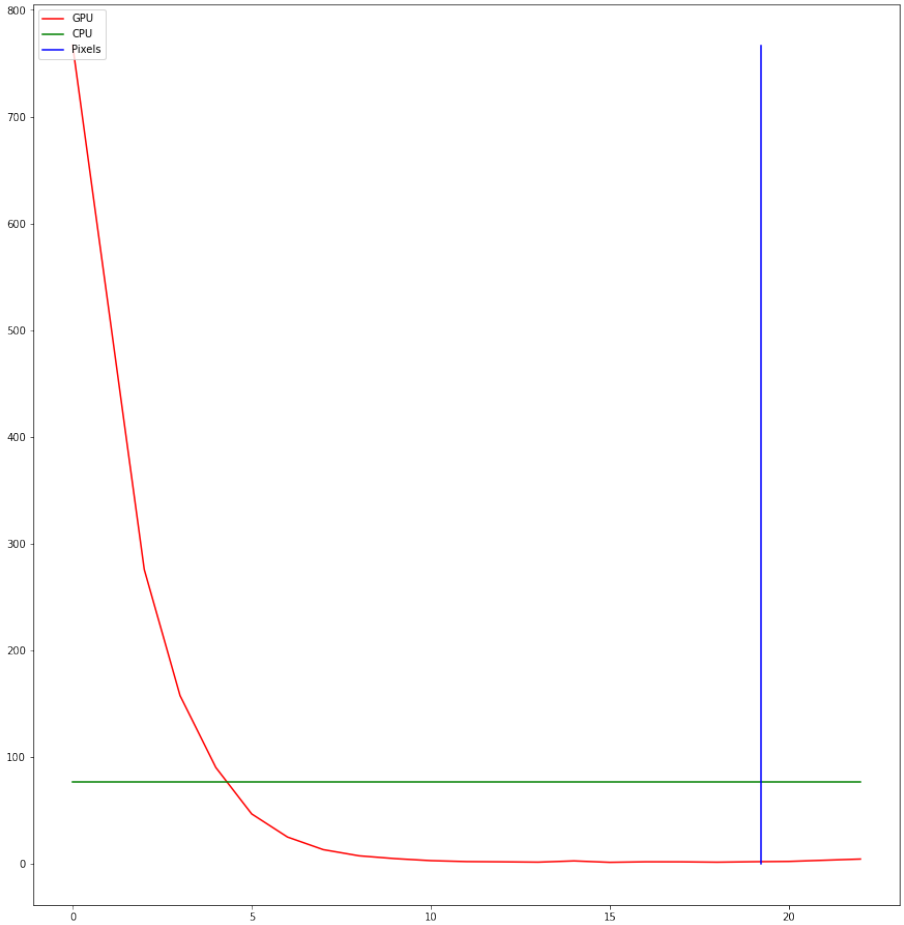


Figure 15: Time execute of Nvidia Tesla A100 vs Intel(R) Xeon(R) CPU @ 2.00GHz. The vertical axis is running time, the horizontal axis is $log_2\left(nThreads\right)$. The green line is CPU time. The red line is GPU time. The blue is $log_2\left(h \times w\right)$

5.2 Number of blocks and number of threads

5.2.1 Nvidia Quadro P620

The Nvidia Quadro P620 is a graphics card with DirectX 12 based on the **Pascal** architecture.

gridDim \ blockDim	**1**	**4**	**16**	**64**	**256**
1	1921.28	538.16	142.68	38.62	12.69
4	497.36	138.78	39.05	10.23	3.68
16	137.35	36.71	10.79	3.97	2.93
64	46.86	13.24	3.93	3.19	3.50
256	36.59	10.92	3.74	2.69	2.42
1024	37.28	9.39	4.18	2.37	2.89

Table 3: Time run on Nvidia Quadro P620.

5.2.2 Nvidia GeForce MX130

The Nvidia GeForce MX130 is a graphics card with DirectX 12 based on the **Maxwell** architecture.

gridDim \ blockDim	**1**	**4**	**16**	**64**	**256**
1	1785.61	509.38	136.54	39.18	13.21
4	464.52	134.97	38.00	12.18	5.80
16	132.12	36.56	11.88	5.67	4.84
64	58.50	17.75	6.82	4.90	4.60
256	52.76	15.54	6.39	4.78	4.37
1024	53.20	15.97	6.18	4.75	4.51

Table 4: Time run on Nvidia GeForce MX130.

5.2.3 Nvidia Tesla T4

The Nvidia Tesla T4 is a graphics card with DirectX 12 Ultimate based on the **Turing** architecture.

gridDim \ blockDim	**1**	**4**	**16**	**64**	**256**
1	1182.08	510.87	199.363	58.71	22.81
4	479.14	190.88	55.50	16.09	7.12
16	187.47	53.25	15.32	5.39	3.30
64	53.0	14.91	5.35	3.22	2.54
256	20.24	6.58	3.05	2.50	2.33
1024	17.36	5.89	2.94	2.39	2.39

Table 5: Time run on Nvidia Tesla T4.

5.2.4 Nvidia Tesla A100

The Nvidia Tesla T4 is a graphics card with DirectX 12 Ultimate based on the **Ampere** architecture.

gridDim \ blockDim	**1**	**4**	**16**	**64**	**256**
1	763.85	275.66	86.30	23.92	7.40
4	286.03	88.73	24.43	7.16	2.92
16	90.35	24.15	7.46	2.97	1.82
64	24.75	7.17	2.94	1.82	1.52
256	7.40	2.89	1.84	1.57	1.49
1024	3.23	1.90	1.59	1.54	1.50

Table 6: Time run on Nvidia Tesla A100.

5.3 NVPROF

Commands for repart memory: *nvprof .*
main images/Chateau.png Chateau_GPU.png 0 256 1 8 1 1 8 8 4. Thence inferred 8 blocks, 25 threads each block; total: 2048 threads.

Commands for normal repart: *nvprof .*
main images/Chateau.png Chateau_GPU.png 0 256 1 8 8 1 8 4 1. Thence inferred 64 blocks, 32 threads each block; total: 2048 threads.

Due to the different number of blocks and threads, the total is still the same. We check other functions and see that the part with more blocks will be 10-20% faster. But functions using shared memory (*_repart_v4_mem*) are faster.

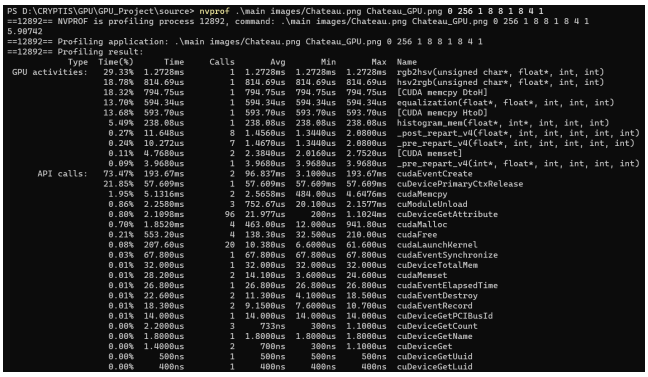


Figure 16: Repart normal. Avg: $3.9680 + 10.272 + 11.648 = 25.888$

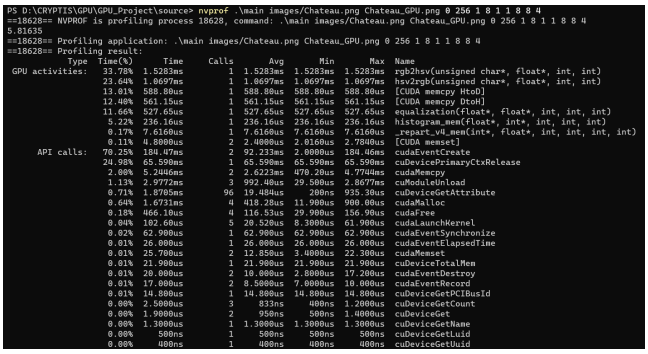


Figure 17: Repart memory. Avg: 7.6160us

5.4 ILP

Implemented in code using while loop.

References

[1] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE transactions on Computers*, vol. 31, no. 03, pp. 260–264, 1982.

[2] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE transactions on computers*, vol. 100, no. 8, pp. 786–793, 1973.

[3] R. Rivest, "Rfc1321: The md5 message-digest algorithm," 1992.