GPGPU - Project report

Nicolas Portal - Alexandre Yvart - Geoffrey Jount

14 juin 2020

Table des matières

1 Introduction

For educational purpose in the GPGPU subject, we are asked to work on a project using a GPU parallel method. We will use the CUDA library.

2 Problem description

This project's goal is to implement an image segmentation algorithm : the max-flow/min-cut algorithm. A picture containing numerous pixels however, processing one after another in a simple `for` loop would show to be very slow. We thus need to make the process parallel, and implement it on GPU for an even faster result : the *push-relabel* algorithm, parallel version of the first mentioned, is what we seek.

3 CPU implementation

We followed the guidelines found on the NVIDIA presentation.
All in all there are eight matrices :
- for the heights of each node
- for the excess flow of each node
- for the links between the source and each node
- for the links between the sink and each nodes

And the four last matrices for the links between each nodes : top, bottom, right and left neighbours matrices were labeled with the inverse of the euclidean distance in RGB space.
The links between the source and each node and the links between the sink and each node were labelled with the inverse of the average distance to the marked pixels in RGB space. Each node were linked to the source and sink. The height of each node was 0 except for the source which had a height equals to the number of nodes. The `std_load` library was used to load `jpg` images both in grayscale and RGB color space. Images were stored as one dimensional vectors with contiguous red, green and blue pixels. the size of these vectors were three times as large as the number of pixels in an image. The `ppm` file format was used to generate the output segmented image.

3.1 Issue 1

Our implementation soon ran into an infinite loop which remained one of the core issue for several weeks. As a result, the code was changed to make it possible for each node to push towards the sink or the source. This did not solved the issue. The weights of links between the source ans sink on one hand and every node on the other hand was changed using the histogram on red, green and blue channel of the marked pixels to compute a probability. This did not solved the issue as well.

It was found that there was actually no infinite loop but the algorithm was very long to terminate as two nodes would often exchange flow between them waiting for one of them to reach the maximum height. To solve this glitch, it was decided to manually lower the maximum height. The maximum height was no longer set to the number of nodes but rather to a low number such as five or ten. This made it possible for our algorithm to terminate.

maxHeight	time
5	146ms
10	271ms
100	1755ms

3.2 Issue 2

An other issue quickly appeared. Using a depth first search to find nodes yielded bad results as every pixel would be set to white.

When it comes to the way to get the output image, it was decided to set every pixel with a height above zero to one. This did not generate optimal results but it was good enough to make out the segmentation.

A small eight by eight image was used to make the debug step simpler. This image was a ppm image with every pixels on the top part of the image set to blue and every pixel on the lower part of the image set to red. One pixel was set to black and white in each region.

3.3 Issue 3

Although the maximum height was decrease to five or ten, the run time was still to high. It took 15 to 20 minutes for the algorithm to terminate.

Lowering the maximum height resulted in a 15, 20 minute running time. This was still too high. It was found out that calling the push and relabel function on every nodes would decrease the run time significantly. The check for active nodes was now done by the push and relabel function. It was no longer needed to write a separate function to retrieve the right active nodes. It was also no longer required to loop on every pixels to find active nodes. This decreased the run time to 10 seconds which was a huge improvement.

3.4 Issue 4

For a long time we simply checked the height value of each pixel to get the min cut of the graph and thus decide the color of each pixel.

Since we were not satisfied with our output segmented image, it was decided to change the way we recovered the min cut. Although there were very little explanation about the way the BFS algorithm should have been implemented in NVIDIA slides, the algorithm was successfully implemented. It was a little bit of a disapointment though since this did not change the look of the output segmented image. This led us to understand that this non optimal segmented image resulted from a flaw in the core algorithm (maybe the weights of the links).

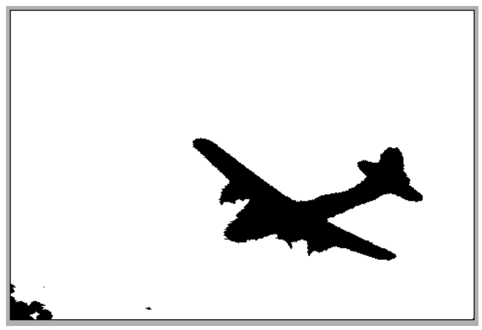
4 optimization

It was found out that the count active function slowed down the algorithm.

In order to make it quicker, It was decided to no longer loop on the whole excess flow matrix but rather, to return true as soon as the function found an active node. This slight change required to change the return type of the function from int to bool. It was not necessary to count every active nodes anymore. The program would keep in the main while loop as long as the count active function returned true. This resulted in a slight but noticeable decrease in run time.

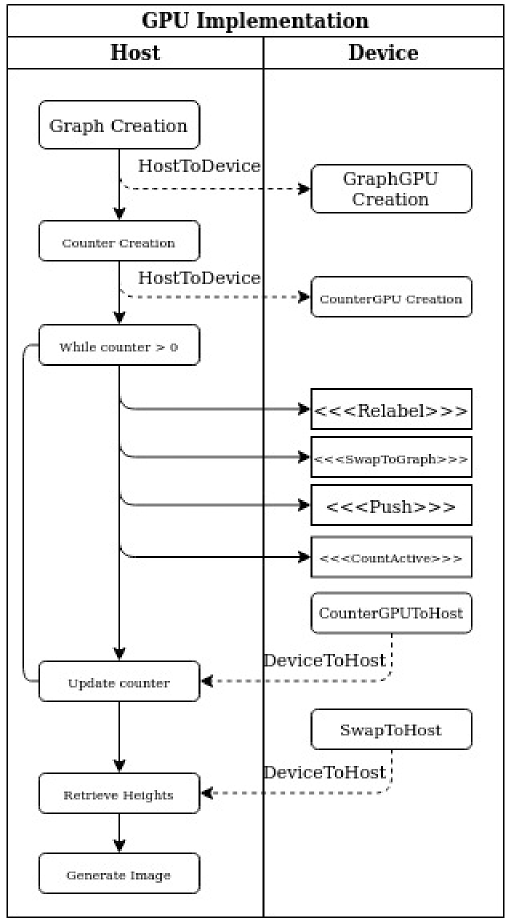
maxHeight = 5	Before	After
Time	159 ms	146 ms

5 Output segmented images


6 GPU implementation

The first GPU implementation is kept as close as possible to the CPU version. The graph structure is instantiated on the CPU just like the CPU version, except that we flattened the matrices on one dimension. The graph structure is then copied on the device, with the addition of a swap matrix used for the relabeling step. Then, the push and relabel kernels are called one after the other, where each thread is representing one node of the graph. The final cut is done just like the CPU version. We didn't have much issues on implementing the GPU version since the CPU version was already functional.



7 Bottlenecks

After proceeding to some profiling, we quickly saw that our kernel used to count_active nodes was a bottleneck because it was taking too much time despite not doing any calculus. Otherwise, most of the bottlenecks are on the allocation of memory on the device which is hard to optimize unless by allocation the whole structure at once. The push kernel is also a bottleneck since it's checking a lot of conditions. Overall, except from reducing the number of calls to cudaMalloc, we can only solve the bottlenecks with algorithmic optimizations.

```
==98771== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   32.93%  2.8255ms        70  40.363us  24.543us  97.596us  push(GraphGPU*)
                   22.75%  1.9516ms        71  27.487us  26.718us  30.046us  count_active(GraphGPU*, int*)
                   22.18%  1.9030ms        70  27.185us  19.167us  48.478us  relabel(GraphGPU*, int*)
                   13.15%  1.1279ms        70  16.113us  15.552us  16.671us  swap_to_graph(GraphGPU*, int*)
                    5.70%  489.20us        19  25.747us  1.1840us  53.310us  [CUDA memcpy HtoD]
                    1.80%  154.62us        72  2.1470us  1.4400us  49.982us  [CUDA memcpy DtoH]
                    1.10%  94.526us        70  1.3500us  1.3440us  1.3760us  [CUDA memset]
                    0.21%  18.111us         1  18.111us  18.111us  18.111us  graph_to_swap(GraphGPU*, int*)
                    0.18%  15.583us         1  15.583us  15.583us  15.583us  setImage(GraphGPU*, int*)
      API calls:   94.21%  218.61ms        12  18.217ms  5.1710us  218.29ms  cudaMalloc
                    4.00%  9.2825ms       292  31.789us  1.2650us  102.25us  cudaDeviceSynchronize
                    0.97%  2.2566ms        91  24.797us  3.5060us  142.50us  cudaMemcpy
                    0.49%  1.1474ms       283  4.0540us  3.2650us  23.019us  cudaLaunchKernel
                    0.09%  201.00us        70  2.8710us  2.6600us  6.0160us  cudaMemset
```

8 Enhancements

8.1 Enhancement 1

We reduced the time spent on the count_active kernel by removing the atomic operations. We didn't need to count how many nodes were active but instead we just had to check if none were active. So, by using a boolean we allowed concurrent accesses to the variable and reduced the time spent on counting nodes by half.

```
==104895== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   39.36%  3.1715ms        70  45.306us  27.358us  98.620us  push(GraphGPU*)
                   23.99%  1.9334ms        70  27.619us  19.551us  48.413us  relabel(GraphGPU*, int*)
                   14.08%  1.1347ms        70  16.209us  15.839us  16.735us  swap_to_graph(GraphGPU*, int*)
                   13.04%  1.0507ms        71  14.799us  12.831us  31.199us  count_active(GraphGPU*, int*)
                    6.06%  488.56us        19  25.713us     544ns  53.278us  [CUDA memcpy HtoD]
                    1.92%  154.94us        72  2.1510us  1.4710us  49.950us  [CUDA memcpy DtoH]
                    1.05%  84.860us        70  1.2120us  1.0560us  1.3760us  [CUDA memset]
                    0.25%  19.775us         1  19.775us  19.775us  19.775us  setImage(GraphGPU*, int*)
                    0.24%  19.168us         1  19.168us  19.168us  19.168us  graph_to_swap(GraphGPU*, int*)
      API calls:   91.37%  139.24ms        12  11.604ms  5.0940us  138.93ms  cudaMalloc
                    5.76%  8.7764ms       292  30.056us  1.2010us  102.67us  cudaDeviceSynchronize
                    1.48%  2.2536ms        91  24.764us  3.2820us  138.10us  cudaMemcpy
                    0.91%  1.3831ms       283  4.8870us  3.0460us  42.539us  cudaLaunchKernel
                    0.13%  201.96us        70  2.8850us  2.6810us  5.0890us  cudaMemset
```

8.2 Enhancement 2

Since most of the time spent by the API calls is on cudaMalloc, we reduced the number of allocation by 1 to gain 10 ms of execution time.

```
==108136== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   37.51%  2.8582ms        70  40.831us  28.895us  88.124us  push(GraphGPU*)
                   26.18%  1.9945ms        70  28.492us  24.063us  44.734us  relabel(GraphGPU*, int*)
                   14.18%  1.0806ms        70  15.436us  14.975us  16.223us  swap_to_graph(GraphGPU*, int*)
                   12.04%  917.59us        71  12.923us  10.975us  30.495us  count_active(GraphGPU*, int*)
                    6.45%  491.05us        19  25.844us  1.1840us  53.726us  [CUDA memcpy HtoD]
                    1.96%  149.34us        72  2.0740us  1.1840us  49.950us  [CUDA memcpy DtoH]
                    1.18%  90.045us        70  1.2860us     928ns  1.3120us  [CUDA memset]
                    0.25%  19.327us         1  19.327us  19.327us  19.327us  setImage(GraphGPU*, int*)
                    0.24%  18.207us         1  18.207us  18.207us  18.207us  graph_to_swap(GraphGPU*, int*)
      API calls:   79.06%  119.78ms        11  10.889ms  4.9440us  119.55ms  cudaMalloc
                   17.77%  26.927ms       292  92.214us  1.2590us  142.54us  cudaDeviceSynchronize
                    1.59%  2.4044ms        91  26.422us  3.5270us  144.97us  cudaMemcpy
                    1.13%  1.7124ms       283  6.0500us  3.2100us  18.558us  cudaLaunchKernel
                    0.15%  221.33us        70  3.1610us  2.7270us  15.903us  cudaMemset
```

9 Benchmarks recapitulation

```
Benchmark                    Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------
BM_cpu/real_time           264 ms          264 ms            3 frame_rate=3.78102/s
BM_gpu/real_time            38 ms           38 ms           19 frame_rate=26.1102/s
```

10 Tasks repartition

Nicolas
- CPU implementation and debug
- Report
- GPU debug

Alexandre
- CPU debug
- GPU implementation

Geoffrey
- CPU debug
- Benchmarking implementation