# WebGL Project : Component Labeling

Martin Binet

February 24, 2018

## 1 Introduction

The software ImageJ[SC12] developed at the National Institutes of Health (NIH), USA, is a Java-based image processing and analysis program. It is available for free, open source, multi-threaded and platform independent, with possibilities of developing plugins to suit specific requirements. One of the most prominent feature of the software is the analysis of particles.

Particle analysis consists of measuring different parameters over objects in the image. It is used in many fields for instance to count and measure cells in biology in an automatic and reliable way. Valuable quantitative information may be collected such as number of cells, shape or fluorescence of the cells. This approach allows to simply compare different population of cells [EC13].

For a binary or thresholded image it is possible to detect automatically the particles of the image and then count and measure them in an automatic procedure. Particle detection identifies each particle in the image. It is specific to binary images. In order to detect each particle independently, they should be distinct. For gray-levels images the process is more difficult to automate, but it opens access to new methods.

To analyze particles, ImageJ performs an analysis of connected components, where each of the pixels constituting the objects will be allocated a label. The labeling algorithm transforms a binary image into a symbolic image in order that each connected component is assigned a unique label. This point is related to the connectivity of the connected components. Various algorithms have been proposed as well[SK03].

Recently, the usage of GPUs (Graphics processing unit) has increased in various fields of informatics. Created at first for the sole purpose of rendering images to a display device from a frame buffer, they prove themselves useful in other applications, such as Deep Learning or protein docking [MMV+10] for instance. GPUs, contrary to CPUs (Central processing unit), are designed to run multiple computations at the same time, and the paradigm change from one to another can be tough to handle. However, with the goal to increase the speed of the programs, propositions have been made about particle detection algorithms specific to GPUs [O'C09].

In this paper we focus our attention on the implementation of two different algorithms for labeling connected components : one aimed at CPU usage, and another one aimed at GPU usage. The two algorithms will first be detailed, and benchmarks will be measured between the algorithms and ImageJ.

## 2 Material & Methods

### 2.1 CPU Algorithm

This algorithm is based on [CCL04]. It scans the image from top to bottom and from left to right. The labels are stored in a labeled image, where particles are labeled starting with 1, and the label -1 is used for some of the background pixels. Before it starts, a dummy row of white pixels has to be added to the top of the image. Let C be the label index, and P be the current pixel. For each black pixel encountered, the steps are as follow :

1. If P is not labeled, and the pixel immediately above is white, then P is an external contour point of a newly encountered component. Label C is then assigned to P, and the `contour tracing` function is called (Step 4). This function, detailed below, will assign the label C to all external contour points of the component.

2. If the pixel right under P is a white unlabeled pixel, then P is an internal contour point. There are two possibilities here :

   (a) P is not labeled. In that case, its left neighbor N must already be labeled, and P gets the same label as N. the `contour tracing` function is then called to label all pixels belonging to that internal contour (Step 4).

   (b) P is already labeled. In that case the `contour tracing` function is simply called without prior modifications (Step 4).

3. If Steps 1 and 2 are false, then the left neighbor N of P must be a labeled pixel. P is assigned the same label as N.

4. `Contour tracing` is a loop that calls the `Tracer` function (Step 5), starting with the initial point P. At the first iteration, it checks if the component is only one pixel in size. If that is the case, the function exits. Otherwise, it means a second pixel S has been found, and it will keep calling `Tracer` until the following conditions hold true :

   (a) `Tracer` outputs the initial point P;

   (b) The contour point following P is S.

5. The `Tracer` function rotates around the pixel until it finds a neighbor, and returns it's position and the angle at which it was found. The rotation matrix used is shown in Fig.1. If the initial position is an external contour point, the search will start at 7, and if it is an internal contour point, it will start at 3. After that, the next search will always start at 2 plus the direction where was the previous pixel (Fig.1). All white pixels encountered by `Tracer` are labeled -1, which prevents from labeling internal contours more than once (as in Step 2).

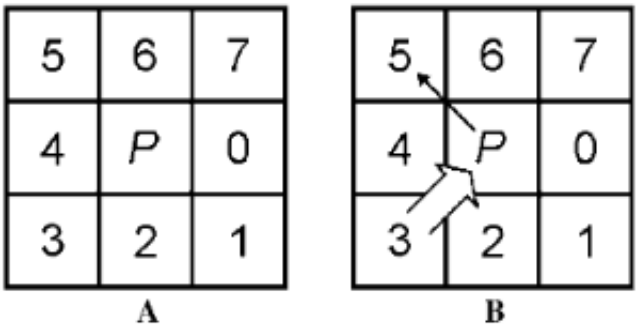


Figure 1: From [CCL04] A) The neighboring points of P are indexed from 0 to 7. (B) If the previous contour point lies at 3, the next search direction is set to be 5.

## 2.2 GPU Algorithm

This algorithm, developed for WebGL, makes use specifically of the fragment shader. There are two main steps here :

1. Each pixel is assigned a label, that is stored in the green and blue canals. The label is its position (x, y). The red label is not modified, thus enabling us to know if it is a white pixel (value of 255) or a black pixel (value of 0). See Fig.2b.

2. For each pixel, all the neighbors in the top, down, right and left directions are checked in order to find the maximum label, and the pixel then takes the same label as the maximum found. For any direction, the search stops when a white pixel is encountered, in order to only check pixels that belong to the component.

The Step 2 is repeated a number of times equal to the height of the image, to ensure that all components are labeled properly (Fig.2c).

(a) (b) (c)

Figure 2: Example of the GPU algorithm. (a) : initial image. (b) : image after the label of pixels (Step 1). (c) : labeled image at the end of the process.

## 3 Results

The two algorithms described above have been compared to the one used by ImageJ. The image used is the one shown in Fig.2, and the benchmark test were run on a computer under linux Debian operating system with a cpu E3-1240 v5, frequency of the cpu of 3.50GHz and with 16 Go of Ram, using Firefox browser for CPU and GPU. The image size was doubled each time until ImageJ or the browser could not handle the memory usage. Results can be seen in Fig.3. For both CPU and GPU algorithms, the size limit was 1 280 000 pixels, while ImageJ could handle much bigger sizes, with a limit at 163 millions of pixels.

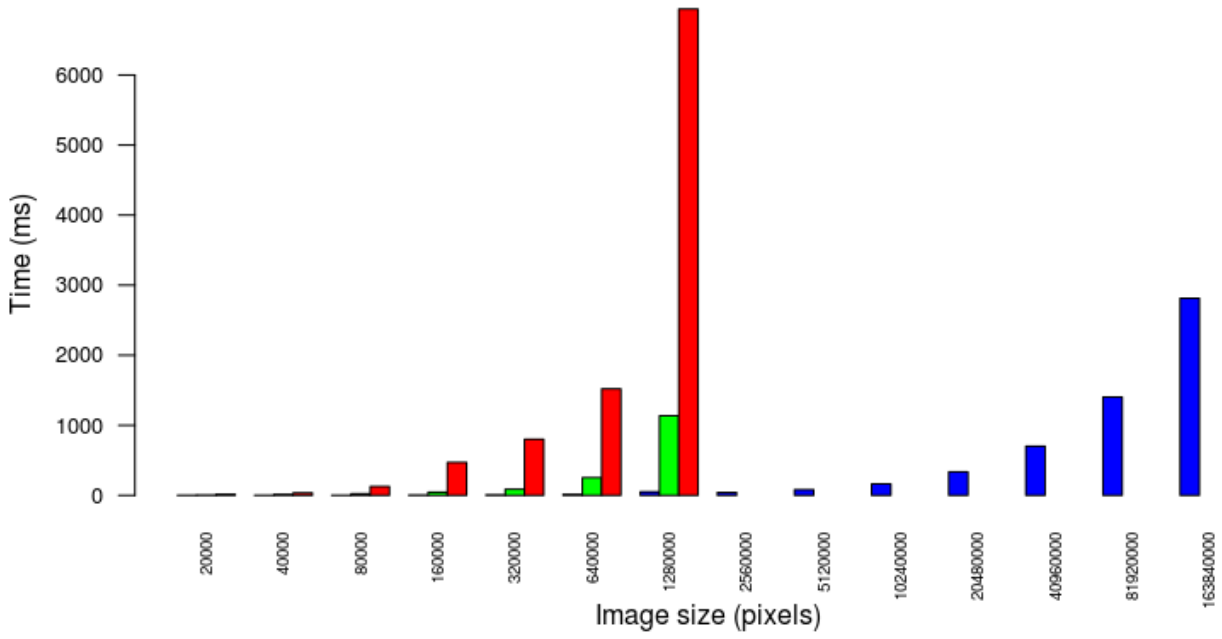


Figure 3: Result of the benchmark test for ImageJ (blue), CPU (green) and GPU (red) algorithms. The image used is the one in Fig.2.

## 4 Discussion

The benchmark results show clearly that both CPU and GPU algorithms are falling far behind the time efficiency of the one present in ImageJ. The CPU algorithm is using a one-pass method, that is it goes over the whole image only once. On the other hand, what is used in ImageJ is a two-pass algorithm, that goes over the whole image twice. In theory, one-pass algorithms are more efficients than two-passes, but in practice this is not always the case, depending on the shape

and the size of the particles. What is more, running the algorithm in the browser is likely slower than with ImageJ, and the Fig.3 shows that there is a maximum size that the browser can handle before crashing. This limit is most likely due to the browser limitations, as it does not handle well high memory usage like ImageJ could.

However, this does not explain why the time skyrockets concerning the GPU algorithm. The probable issue here is the multitude of `for` loops in the fragment shader. Other implementations, such as [O'C09], solve this by using the vertex shader as well, and by partitioning the process into several more steps, which are run sequentially. The next step for improving the GPU algorithm would be to include computations within the vertex shader. Here the computation was done solely in the fragment shader, and this way is rarely appropriate. It is a much better practice, indeed, to break the program between vertex and fragment shader.

Additionally, more benchmark tests could have been run with different images and more shapes. Results for a given environment do not imply it will be similar in other environments. Using more images could have helped pinpoint the weaknesses of the algorithms.

# 5 Conclusion

GPU usage to analyze images is still relatively new, and current algorithms still have issues in edge cases. Nevertheless, the gain in speed is always a considerable asset. In this paper, the proposed GPU algorithm is far from the current standard, and still needs a lot of refinement. A lack of understanding concerning bits encryption, the way the vertex shader operates and the higher API at our disposal was an obstacle to the development of the algorithm.

# References


[CCL04] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. A linear-time component-labeling algorithm using contour tracing technique. *computer vision and image understanding*, 93(2):206–220, 2004.

[EC13] Jensen EC. Quantitative analysis of histological staining and fluorescence using imagej. *Anat Rec (Hoboken)*, 296(3):378–381, 2013.

[MMV+10] Gary Macindoe, Lazaros Mavridis, Vishwesh Venkatraman, Marie-Dominique Devignes, and David W Ritchie. Hexserver: an fft-based protein docking server powered by graphics processors. *Nucleic acids research*, 38(suppl_2):W445–W449, 2010.

[O'C09] Sean M O'Connell. A gpu implementation of connected component labeling. *Illinois State University*, 2009.

[SC12] Eliceiri KW. Schneider CA, Rasband WS. Nih image to imagej: 25 years of image analysis. *Nat Methods.*, 9(7):671–675, 2012.

[SK03] Sugie NO Suzuki KE, Horiba IS. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89:1–23, 2003.