ESIR-SPP Lab 3-4: Multi-threaded Image Filtering

François Taiani (ftaiani.ouvaton.org)*

1 Objective of the lab

The objective of this lab is to implement a simple multi-threaded image filtering engine and to analyze its performance.

2 Preliminaries

First check that your lab pair is properly registered on the Moodle website of the course.

Download the lab archive on the course website. Compile and execute `SimpleImageProcessingExample.java`. This program reads the image `TEST_IMAGES/15226222451_5fd668d81a_c.jpg` and produces the image `tmp.png` in which the red, green, and blue components of each pixel have been rotated.

Look at the code of `SimpleImageProcessingExample.java`, and familiarize yourself with the classes `java.awt.image.BufferedImage`, `javax.imageio.ImageIO`, and `java.awt.Color`. For instance, try to modify the program to only keep the red component of each pixel.

3 Part I (3 points): single threaded image filtering engine

In this part, you are asked to implement a single-threaded image filtering engine in Java. Your program should conform to the two interfaces `IImageFilteringEngine` and `IFilter`, whose source is provided in the lab archive.

3.1 The interfaces `IImageFilteringEngine` and `IFilter`

`IImageFilteringEngine` specifies the main interface of your filtering engine.

```
public interface IImageFilteringEngine {
  public void loadImage(String inputImage) throws Exception ;
  public void writeOutPngImage(String outFile) throws Exception ;
  public void setImg(BufferedImage newImg) ;
  public BufferedImage getImg() ;
  public void applyFilter(IFilter someFilter) ;
} // EndInterface IImageFilteringEngine
```

The two methods `loadImage(..)` and `setImg(..)` are used to either load or set a new working image in the filtering engine. The method `applyFilter(..)` applies a filter (passed in parameter) to the current image stored by the filter. Finally the methods `getImg()` and `writeOutPngImage(..)` are used to obtain a reference to or, respectively, save to a PNG file the current image of the engine.

A typical usage of the interface `IImageFilteringEngine` would be as follow:

```
IImageFilteringEngine im = new MyImageFilteringEngine();
im.loadImage("./TEST_IMAGES/15226222451_5fd668d81a_c.jpg");
im.applyFilter(new GrayLevelFilter());
im.applyFilter(new GaussianContourExtractorFilter());
im.writeOutPngImage("./TEST_IMAGES/15226222451_5fd668d81a_c_gaussian_contour.png");
```

*francois.taiani@irisa.fr

The above code creates a new `MyImageFilteringEngine` object, loads the image `15226222451_5fd668d81a_c.jpg`, applies two filters to this image (`GrayLevelFilter` and `GaussianContourExtractorFilter`), and finally writes out the result to the file `15226222451_5fd668d81a_c_Gray_Contour.png`.

A filter passed to `applyFilter(..)` should implement the interface `IFilter` listed below.

```
public interface IFilter {
  public int  getMargin();
  public void applyFilterAtPoint(int x, int y,
                               BufferedImage imgIn,
                               BufferedImage imgOut);
} // EndInterface IFilter
```

`getMargin()` should return the number of neighboring pixels in all four directions that the filter requires to compute a pixel value in the new image. As a result the size of the image produced by the filter will generally be smaller than the original image. More specifically if the original image has a dimension of $w \times h$ pixels, and if `getMargin()` returns m, then the new image should contain $(w-2m) \times (h-2m)$ pixels.

`applyFilterAtPoint(..)` should apply the filter around the pixel (x, y) in the input image `imgIn`, and set the pixel $(x-m, y-m)$ of `imgOut`, where m is the margin returned by `getMargin()`. During this computation, the filter should only use pixel values located in a square bordered by the $(x-m, y-m)$ and $(x+m, y+m)$ positions of the input image.

3.2 Single-threaded filtering engine, and gray and contour detection filters

In this first part of the lab you are asked to implement two filters: one that produces a gray-scale version of an image (margin = 0), and one that implements a simple contour detection algorithm, described below. Together with these two filters, you should implement a single-threaded version of the filter engine called `SingleThreadedImageFilteringEngine`, that implements the interface `IImageFilteringEngine`.

3.2.1 Gray Filter

A $w \times h$ raster image i can be modeled as a function $i : [0..w-1] \times [0..h-1] \mapsto \mathcal{C}$, which maps each pixel position $(x, y) \in [0..w-1] \times [0..h-1]$ to a color $i(x, y) \in \mathcal{C}$, where $\mathcal{C}$ denotes the set of all possible colors. We will work in the following with RGB pixels (in which each color is encoded by a combination of red, green, and blue values, each in the range `0x00` to `0xFF`). For simplicity's sake, we will note $i(x, y).r$, $i(x, y).g$ and $i(x, y).b$ the red, green, and blue value of the pixel (x, y) of image i.

Your gray filter (called `GrayLevelFilter`) should generate an output image o from an input image i so that the red, green and blue values of o's pixels are the average of the red, green and blue values of i's. This transformation can be expressed with the following formula:

$$o(x, y).r = o(x, y).g = o(x, y).b = \frac{1}{3}\Big(i(x, y).r + i(x, y).g + i(x, y).b\Big)$$

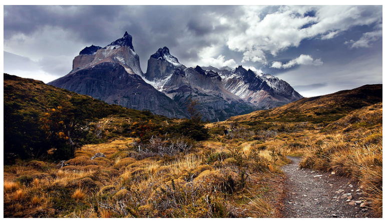

Figure 1: Original image

Figure **??** shows the result of applying the `GrayLevelFilter` on the picture of Figure **??**.

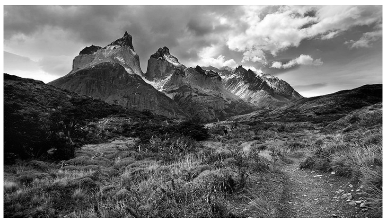

Figure 2: Result of the `GrayLevelFilter` on the picture of Fig. ??

3.2.2 Gaussian Contour Extraction

You are now asked to implement a Gaussian version of Canny's edge detector[1] for your contour extraction filter, which you will call `GaussianContourExtractorFilter`. You will assume that the original image i you work on is already a gray-scale image, and you will work on the blue component of pixels in the following (since all components should have the same value).

Your filter should for each pixel of the original image i compute an horizontal and vertical gradients Δ_x and Δ_y, smoothed using a Gaussian-weighted average over a 11×11 square around the pixel of interest:

$$\Delta_x = \sum_{(d_x,d_y)\in[\![-5,+5]\!]} \operatorname{sign}(d_x) \times i(x+d_x, y+d_y) \times \exp\left(-\frac{1}{4} \times \left(d_x^2 + d_y^2\right)\right),$$

$$\Delta_y = \sum_{(d_x,d_y)\in[\![-5,+5]\!]} \operatorname{sign}(d_y) \times i(x+d_x, y+d_y) \times \exp\left(-\frac{1}{4} \times \left(d_x^2 + d_y^2\right)\right),$$

where $[\![-5,+5]\!]$ is the integer interval ranging from -5 to $+5$, and $\operatorname{sign}(x)$ returns the sign of x, i.e. -1, $+1$, or 0, depending whether x is negative, positive, or null, respectively.

You should then use Δ_x and Δ_y to compute the norm of the gradient at pizel (x, y), noted $|\vec{\nabla} i(x, y)|$:

$$|\vec{\nabla} i(x,y)| = \sqrt{\Delta_x^2 + \Delta_y^2}.$$

Finally, for a visually pleasant result, the resulting pixel in the output image should be put to a gray value of

$$\max\left(0, 255 - \frac{1}{2} \times |\vec{\nabla} i(x,y)|\right).$$

In terms of Java types, use `Double` variables for all computations, and use a simple final cast to `int` for the final value.

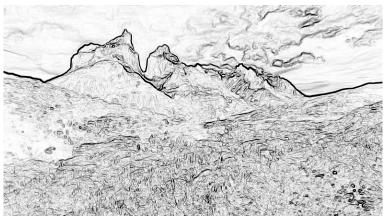

Figure 3: Result of the `GaussianContourExtractorFilter` on the picture of Fig. ??

Figure ?? shows the result of applying the `GaussianContourExtractorFilter` on the gray scale picture of Figure ??.

3.2.3 Testing your filter engine

You should implement a JUnit test set that takes a class implementing `IImageFilteringEngine`, and tests that this image filtering engine is able to execute the filter `GrayLevelFilter` and `GaussianContourExtractorFilter` correctly.

[1] `https://en.wikipedia.org/wiki/Canny_edge_detector`

Your test set should include an application of the above filters on the following images:

- A 256×256 white rectangle ;
- A 256×256 black rectangle ;
- 256×256 rectangles filled with red, green and blue respectively ;
- The provided images `FourCircles.png` and `15226222451_5fd668d81a_c.jpg`, whose expected output images are provided in the files `FourCircles_Gray.png`, `FourCircles_Gray_Contour.png`, `15226222451_5fd668d81a_c_Gray.png` and `15226222451_5fd668d81a_c_Gray_Contour.png`, respectively.

You are encouraged to adopt a test-driven approach[2] for your development, in which you first implement empty skeletons of your implementation classes, before writing your tests, and only then implementing your application.

4 Part II (10 points): multi-threaded image filter

You are now asked to implement a multi-threaded version of your filtering engine, called `MultiThreadedImageFilteringEngine`. The number of worker threads k used by your engine should be passed to the constructor. Because threads are costly to create, you should only create the worker threads of your engine once, in the engine's constructor, and the same threads should be reused each time you call `applyFilter(..)`.

Conceptually your new `applyFilter(..)` method should therefore perform the following steps:

1. distribute work among the k worker threads ($*$);
2. unblock the k worker threads (using the appropriate action);
3. wait for all worker threads to complete their iteration ($**$).

Notes:

- There are 2 points of synchronisation in this code: The worker threads must wait for the main thread to initialise the work to be done ($*$); and the main thread must then wait for the worker threads to finish their share of the work ($**$). In both cases, you will need a shared synchronisation object to implement the synchronisation. You might want to use `CyclicBarrier` objects.
- The output image will be modified concurrently by all worker threads. Do you need to protect it using a lock or a monitor? (You might want to read about memory barriers, and the memory consistency guarantees of `CyclicBarrier` objects.)
- To distribute work among the worker threads, you essentially need to tell each of them which part of the image to work on. Do this by having each thread store the coordinates of the area it should process in appropriate attributes of your Thread or Runnable subclass. Then, have the main thread set this range for each worker thread before unblocking the worker thread.
- Use the interruption mechanism to terminate all the workers threads when terminating your program. You will need to write an appropriate try-catch block in your implementation of worker threads.

Once you have implemented a parallel version of your filtering engine, **check this version with the JUnit tests developed in the previous part for different values of k** ($k \in \{1, 2, 4, 8, 16, 32\}$).

[2] `https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/`

5 Part III (4 points): performance analysis

The goal of this final task is to compare the performance of your sequential and parallel versions of your filtering engine.

You should run two experiments:

1. In a first experiment, measure the time taken to apply the two filters `GrayLevelFilter` and `GaussianContourExtractorFilter` to the image `15226222451_75d515f540_o.jpg` with
   - Your original sequential version;
   - Your parallel version with the number of threads varying from 1 to 10.

   Be sure to measure the execution time of each filter separately. For each filter, chart the results on a graph (one graph per filter). You should run your experiments multiple times to obtain representative averages. (Would you know how to compute a confidence interval on these values?)
2. Repeat the same measurements for both filters with the pictures `15226222451_*.jpg` and $k = 4$ threads. Again, chart the resulting executing times of each filter against the size of the processed images (in pixels). How do you interpret your results?

Each of the four charts earns you up to 0.5 point (for a total of 2 points), and the explanation of each chart earns you up to 0.5 point (also for a total of 2 points).

6 Deliverable and Marking Scheme

6.1 Deliverables

- Source code
- Report of maximum 1000 words (one collective report for the group): As above, your report should describe the overall working of your code and set-up, your results, and comment these results (why you think you obtained such results). Your report should also highlight any technical challenge or limitation you have encountered.

6.2 Marking

You need to demonstrate your solution to obtain a mark. **A submitted solution that is not demonstrated will receive a grade of zero.** As soon as you have completed a part, you can ask for it to be marked. You should submit your code and report using moodle. The last lab session will be used for marking only. Be sure to submit your submissions the day before at the latest (see Moodle for details).

6.3 Marking scheme

- Part I: 3 points
- Part II: 10 points broken down as
  - Your parallel version passes your JUnit tests (provided these tests are correct): 2 points
  - You are able to explain how your parallel version works: 3 points
  - You have properly managed and synchronized your worker threads: 3 points
  - Code structure, clarity and readability: 2 points
- Part III: 4 points broken down as
  - Results varying the number of worker threads: 2 points
  - Results varying the image sizes: 2 points

- Report: 3 points

— End of the lab brief —