Rank filters: variance filter

AUTHORS : *Franck Soubès

1.Introduction

Since image have been digitized on a computer's memory, it has been possible to interact in new ways with those images that were otherwise impossible. This is called image processing, and it consist of methods used to perform operations on a digital image. Those methods are described with various algorithms that can be used for various purposes in multiple fields. Those applications are used for noise filtering and other image enhancement as well as extracting information from images. In this project, will be examined the algorithms used for three types of 2D rank filters : median, min&max and variance.

By definition rank filters are non-linear filters using the local gray-level ordering to compute the filtered value[1] . The output of the filter is the pixel value selected from a specified position in this ranked list. The ranked list is represented by all the grey values that lies within the window which are sorted, from the smallest to the highest value. For an identical window the pixel value will differ in function of the filters used (median, min, max and variance). Moreover the size of the window is also influencing the output pixel. The variance filter is used to edge detection. Edges can be detected using the 1st (Sobel or Cany approaches[2][3]) or 2nd deriviates(Log approach[4]) of the grey level intensity. Nevertheless there's other alternatives using synthetic and real images with the variance filter[5] or simply with a simple formula. The three filters have their own field of expertise. They can be used for removing noise (median filter), detecting edge (variance filter) and mathematical morphology(min/max filters). The main issues of these filters algorithm are their slowness, to overcome these problems the use of small windows and/or low resolution images is required[6]. The last part of this project include an implementation of our algorithm with Webgl[7] in order to highly increase the performance of the algorithm while using the GPU.

In this report, we shall begin by describing the main algorithm with the cpu and then explain how to implement it in webgl . * Variance filter

Next step will be to perform a benchmark on different imageJ plugins, with the objective of comparing their performances such as execution time and the memory load for the Java Virtual Machine (JVM). The ImageJ plugins compared are the default ImageJ RankFilters plugin which has implementation for the median, maximum, minimum and variance filters.

[1]Soille P. On morphological operators based on rank filters. 2002;35(2):527–535. Pattern recognition.

[2]Canny J. A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence. 1986;(6):679–698.

[3]Kittler J. On the accuracy of the Sobel edge detector. 1983;1(1):37–42. Image and Vision Computing.

[4]Marr D, Hildreth E. Theory of edge detection. Proceedings of the Royal Society of London B: Biological Sciences 1980;207(1167):187–217.

[5]Fabijańska A. Variance filter for edge detection and edge-based image segmentation. In: Perspective Technologies and Methods in MEMS Design (MEMSTECH), 2011 Proceedings of VIIth International Conference on. IEEE; 2011. p. 151–154.

[6]Weiss B. Fast median and bilateral filtering. 2006;25(3):519–526. Acm Transactions on Graphics (TOG).

[7]Vasan SN1, Ionita CN, Titus AH, Cartwright AN, Bednarek DR, Rudin S; Graphics Processing Unit (GPU) implementation of image processing algorithms to improve system performance of the Control, Acquisition, Processing, and Image Display System (CAPIDS) of the Micro-Angiographic Fluoroscope (MAF), Proc SPIE Int Soc Opt Eng.

2.Material & Methods

In image processing, variance filter is often used for highlighting edges in the image by replacing each pixel with the neighbourhood variance.

$$s^2 = \frac{\Sigma X^2 - \frac{(\Sigma X)^2}{N}}{N-1}$$

Equation_1: Variance filter is computed here by subtracting the sum of the pixel square with two times the sum divided by the number of pixels in the kernel and overall divided by the number of pixels - 1 this method correspond to the naive algorithm.

This filter is implemented in imageJ through the class rankfilters in Java. For variance algorithm, according to the input image and the size of the kernel, it will not react in the same way. If the kernel's radius size is less than 2 (5x5), it will compute the sum over all the pixels, whereas for a kernel's radius size greater than 2, the sum won't be calculated. In that case this sum is calculated for the first pixel of every line only. For the following pixels, it'll add the new values and subtract those that are not in the sum any more. This way, the computational time is then reduced. Once, the kernel reaches the end of the thread, it start over at the next line until the end of the input image. It's notable that the variance algorithm is closely related to the mean algorithm. In application, this algorithm works by using one "window" defined here by a circular kernel, which slides, entry by entry until the end of the signal. It can process through rows or columns. This method is simple, moreover it's characterised by low computational complexity compared to other methods (Cany, Sobel). However it's not devoid of weakness because of its low resistance to noise. Indeed the impulse and Gaussian noise significantly decreases quality of edge detection [8].

Naive algorithm

The basic method describe here, use a kernel that can be of different types (circular, square, diamond …) and slide over the pixels by modifying them one by one. By definition the modifying pixel is always the central pixel. Hence, a kernel is generally always represented by odd values (3x3,5x5). In the field of the imagerie this process is referred to the convolution[ˆKRU1994] and it's widely use for image processing. The example below explain how a convolution mask operate over an image.

[8]Fabijańska A. Variance filter for edge detection and edge-based image segmentation. In: Perspective Technologies and Methods in MEMS Design (MEMSTECH), 2011 Proceedings of VIIth International Conference on. IEEE; 2011. p. 151–154.

a) Consider a matrix $M = \begin{bmatrix} 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 3 & 4 & 4 \\ 1 & 1 & 5 & 6 & 6 \\ 1 & 1 & 5 & 6 & 6 \end{bmatrix}$

b) In this example we're using a small kernel of size 3 by 3 and we're not considering boundary issues, starting from matrix M(1,1) place the "Window", the value to be changed is the middle element 1, $Kernel = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 2 \\ 1 & 1 & 3 \end{bmatrix}$

c) Use the formula to compute the new value, variance = 0.47(ū = 1.44, **σ** = 0.68)

Change the value of 1 by 0.47 (output image)

d) The procedure is then repeated by sliding the window to the next position M(1,2)

$Kernel = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \\ 1 & 3 & 4 \end{bmatrix}$ Pixel by pixel until the end.

e) The output matrix is $M = \begin{bmatrix} 0.47 & 1.06 & 1.43 \\ 1.73 & 3.33 & 3.36 \\ 2.75 & 4.02 & 1.11 \end{bmatrix}$

Integral image algorithm

The Integral Image is used as a quick and effective way of calculating the sum of values (pixel values) in a given image – or a rectangular subset of a grid[9]. The method for variance filtering make use of a faster algorithm to compute the variance of the pixels in a window[10][11]. From a starting image I, compute an image I' for which the pixel I'(x,y) take as value the sum of all pixels values in the original image between I(0,0) and I(x,y) included [Fig. 2].

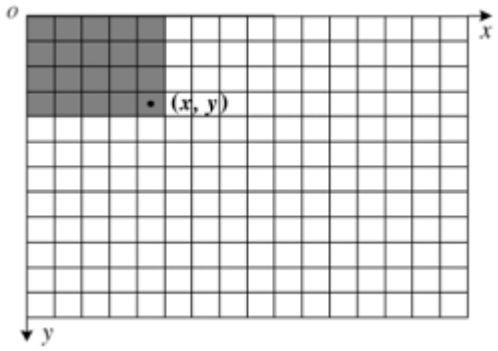

Fig 2. An example of the area pixels between I(0,0) and I(x,y) included.

Afterwards compute an image I'‘for which the pixel I"(x,y) take as value the sum of all squared pixels values in the original image between I(0,0) and I(x,y) included. Here is an example[Fig. 3] of what the function should do for both input images.

$$I = \begin{bmatrix} 5 & 2 & 5 & 2 \\ 3 & 6 & 3 & 6 \\ 5 & 2 & 5 & 2 \\ 3 & 6 & 3 & 6 \end{bmatrix}$$

$$I' = \begin{bmatrix} 5 & 7 & 12 & 14 \\ 8 & 16 & 24 & 32 \\ 13 & 23 & 36 & 46 \\ 16 & 32 & 48 & 64 \end{bmatrix} \quad \begin{bmatrix} 25 & 29 & 54 & 58 \\ 34 & 74 & 108 & 148 \\ 59 & 103 & 162 & 206 \\ 68 & 148 & 216 & 296 \end{bmatrix} = I''$$

[9]Shivani Km, A Fast Integral Image Computing Methods: A Review Design Engineer, Associated Electronics Research Foundation, C-53, Phase-II, Noida (India)

[10]Viola P, Jones M. Rapid object detection using a boosted cascade of simple features. In: Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. vol. 1. IEEE; 2001. p. I–I.

[11]Sarwas G, Skoneczny S. Object localization and detection using variance filter. In: Image Processing & Communications Challenges 6. Springer; 2015. p. 195–202.

Fig 3. An exemple of a image I and the new computed image I' and I".

To compute the variance for a window B on the original image bounded by the coordinates(x,y,w,h), where x<=w and y<=h, compute :

$$I'(B) = I'(x-1, y-1) - I'(x+w, y-1) - I'(x-1, y+h) + I'(x+w, y+h)$$

and

$$I''(B) = I''(x-1, y-1) - I''(x+w, y-1) - I''(x-1, y+h) + I''(x+w, y+h),$$

where I'(x,y) is the sum of all pixels values between I(0,0) and I(x,y) inclusive and I"(x,y) is the sum of all squared pixels values between I(0,0) and I(x,y) inclusive. The variance of the pixels value in the window B is :

$$\sigma^2 = \frac{1}{n} I''(B) - [\frac{1}{n} I'(B)]^2$$

Choice of algorithm for webgl implementation

The naive algorithm is used for computing the variance with the convolve and also for the webgl implementation. The main reason to use this algorithm is that it can be computed in one pass. The method with the convolve was implemented by JC Taveau, while I implemented an other method based on integral image[12][13] both of these methods are available in TIMES(CPU). However, the previous method is not that performant as described in the benchmark and it's rather difficult to implement it in webgl. Hence, the one pass algorithm was mainly picked for his performance and it's the same algorithm than ImageJ.

Webgl implementation

Webgl for Web Graphics Library is based on JavaScript language and dispose of an API very detailed (khronos) and it is mainly used to display interactive 2D or 3D graphics. It is compatible with all the common web browser without the use of any-plugins. In order, to implement the variance filter, we're using the kernel build by JC Taveau. For a given size, the kernel act as an object containing various coordinates of offset depending on the axis x or y. For a kernel of 3x3 it will then contain 9 coordinates. Thus, from the central pixel it's possible to determinate and access to all the neighbouring pixels.

Based on this information, we map over those coordinates in order to store (horizontalOffset and verticalOffset) and use them in the fragment shader. The fragment shader use parallelism to iterates through the textures in the aim of set color values for the textures depending on the process. Hence, the performance is only depending on the hardware and the implementation. The more the GPU has core the more pixels will be treated in parallel by multi-threading. The pixel values within the fragment shader have to be in a range of [0...1]. The coordinates are divided by the width for horizontal offset and by the height for the vertical offset.

[12]Bradley D, Roth G. Adaptive thresholding using integral image. Journal of Graphics Tools. Volume 12, Issue 2. pp. 13-21. 2007. NRC 48816.

[13]Sarwas G, Skoneczny S. Object localization and detection using variance filter. In: Image Processing & Communications Challenges 6. Springer; 2015. p. 195–202.

The main strength of the naive algorithm, is that the variance can be computed in a single pass within a single loop. The algorithm of such method is described down below with the pseudo code.

```
Let n ← 0, Sum ← 0, SumSq ← 0
For each pixels x:
  n ← n + 1
  Sum ← Sum + x
  SumSq ← SumSq + x × x
Var = (SumSq - (Sum × Sum) / n) / (n - 1)
n <- number of pixels for a given kernel
```

The same process is realized in the fragment shader with the computation of the sum, the square sum, then the subtraction of those two and put the result in the output color. The main issues here was to pass those values that are between 0 and 1 because of the fragment shader treat only values in a set of range between [0..1] as said earlier. For a raster type of 8 bit we multiply the pixels containing in the output color by 255. As a result, we're making sure that there's no value over 255 and so for the other types of raster. However for the 16 bit, we weren't able to find a solution to find a threshold that's fit to display the image with variance filter probably due to the the texture gl.REDUI16.

In order to find the perfect threshold, I used the step method proposed by the edge detection group with their agreement, however it was not concluant. Indeed, this method transform pixels to binary values but when filtering with variance there's a specter of grey pixels, moreover the execution time was two times slower than without the tresholding. This code can be seen in the gpuVariance.js in commentary

Benchmarking analysis

Benchmarking analysis is a method widely used to assess the relative performance of an object[14]. That way, it's possible to compare the performance of various algorithms. Only execution time and memory load will be analyzed here. In order to perform this benchmark, one script was implemented. The first script, named *benchmark2* whose aim is to compute the time speed between the start and the end of an input image coming from ImageJ during the filtering process. This script was implemented using the ImageJ macro language. The operation process is run 1000 times for ImageJ measurements to provide robust data. In order to not recording false values we're not considering the first 100 values. Indeed during the execution, we must take into account the internal allocations of the loading images which may introduce error in our measurement. For our own algorithm we did only 50 iterations because of the amount of time that each algorithm takes.

For this project the benchmark was performed with the operating system Linux (4.9.0-3-amd64) using the 1.8.0_144 version of Java and running with the 1.51q version of ImageJ. The model image of this benchmark is Lena for various pixels size for the GPU the graphical card is a nvidia 1050 ti.

[14]Philip J. Fleming and John J. Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM 29, 3 (March 1986), 218-221.

3.Results

Image result comparison between ImageJ, CPU and GPU

Variance filter

The following figure shows the result of our *variance* function for a boat of 720x576 pixels taking as parameter a kernel of diameter = 3 compared to the variance filter of ImageJ with a kernel radius = 1. The results are different for mainly two reasons one is that the luminosity between B and C are different and the fact that the kernel on ImageJ is circular whereas our kernel is square.

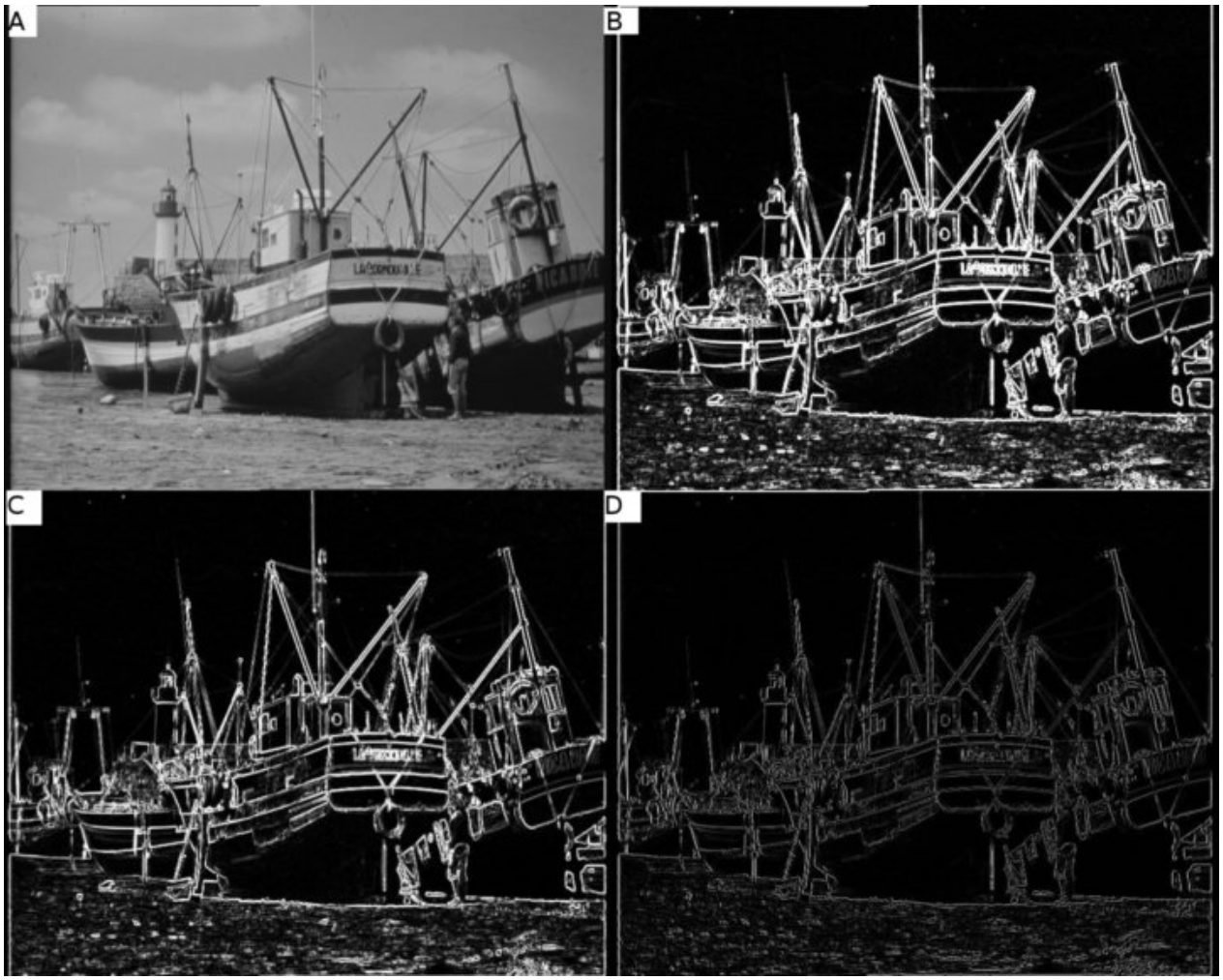

Fig 9. Result of a variance processing for the method based on integral image with (A) representing the original image, (B),(C) and (D) are respectively corresponding to the ImageJ

***variance*, our *variance* function and the subtraction of those two (B-C).**

The following figure shows the result of J.C Taveau variance (One-pass algorithm) for a boat of 720x576 pixels taking as parameter a kernel of diameter = 3 compared to the variance filter of ImageJ with a kernel radius =1. The results are different for mainly two reasons one is that the luminosity between B and C are different and the fact that ImageJ adjust the brightness and contrast of the image after the process

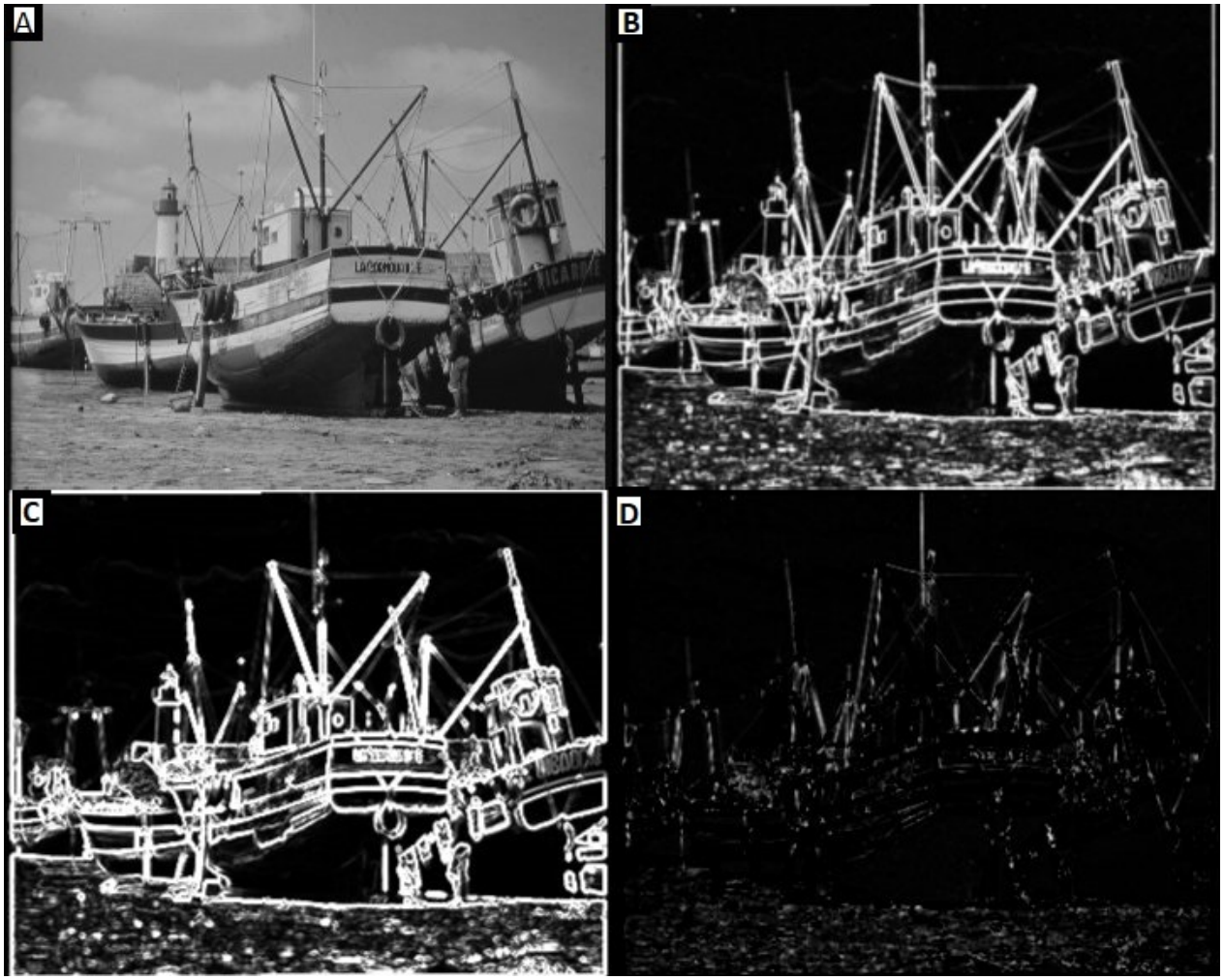

Fig 9. Result of a variance processing for the One-pass algorithm (CPU) with (A) representing the original image, (B),(C) and (D) are respectively corresponding to the ImageJ *variance*, our *variance* function and the subtraction of those two (B-C) for a circular kernel of radius = 1.

The following figure shows the result of our GPU implementation of variance (One-pass algorithm) for a boat of 720x576 pixels taking as parameter a kernel of diameter = 3 compared to the variance filter of ImageJ

with a kernel radius = 1.The results are different for mainly two reasons one is that the luminosity between B and C are different and the fact that ImageJ adjust the brightness and contrast of the image after the process. Moreover, we obtain the same result between the GPU and CPU based on the substraction that's logic because the variance is computed with the same algorithm.

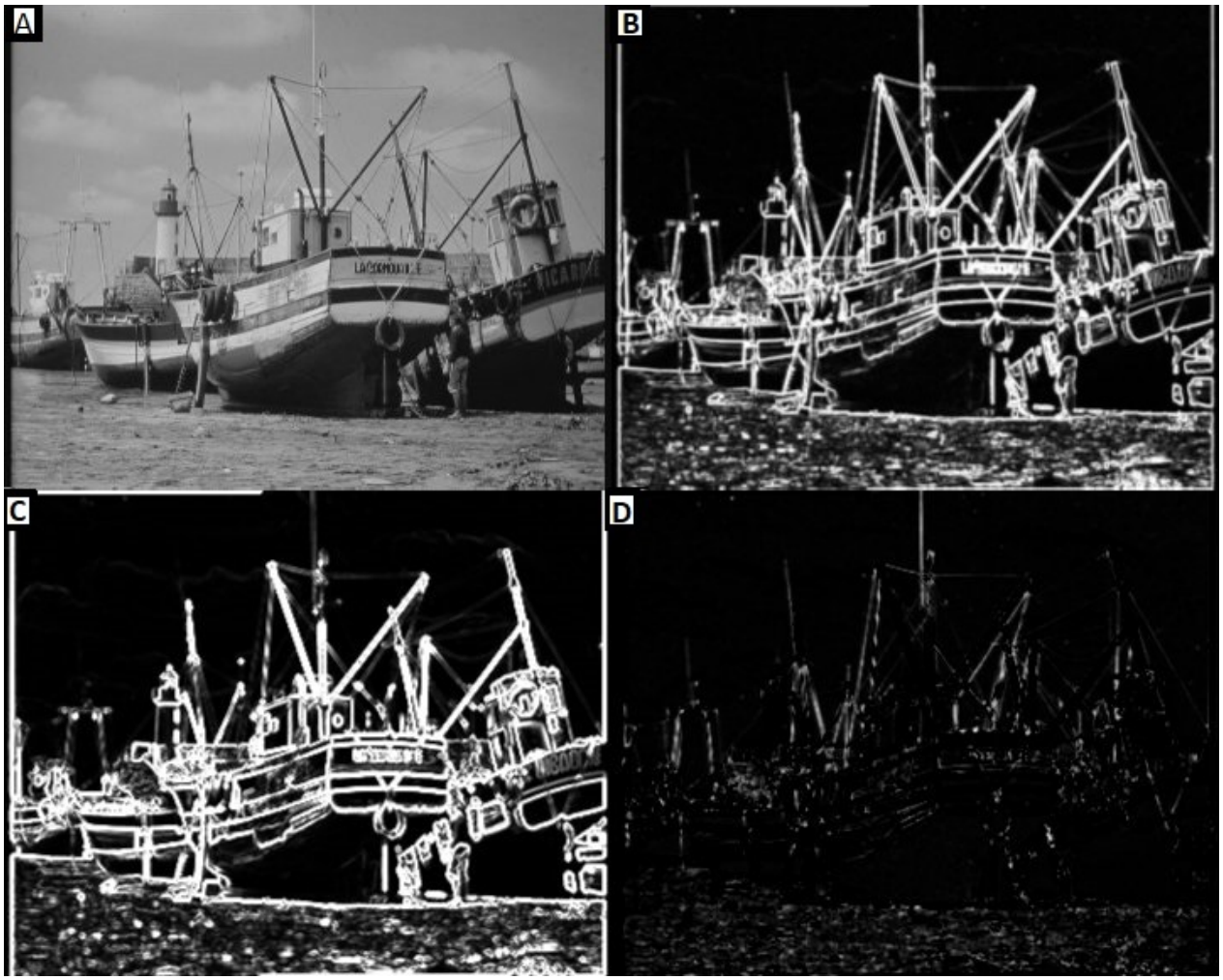

Fig 9. Result of a variance processing for the One-pass algorithm (GPU) with (A) representing the original image, (B),(C) and (D) are respectively corresponding to the ImageJ *variance*, our *variance* function and the subtraction of those two (B-C) for a circular radius = 1.

Benchmark comparison between ImageJ, CPU and GPU for the variance filter

Benchmark integral image vs one pass algorithm for a kernel radius = 1

A comparative benchmark for our own Variance filter based on integral image against the Variance filter based on a single has been done with a set of 7 images for seven different resolution 360x288, 720x576, 900x720, 1080x864, 1440x1152, 1880x1440 and 2880x2304. Each set of 3 images have the same resolution but with a different type, either 8bit, 16bit or float32. The benchmark representation is represented down below :

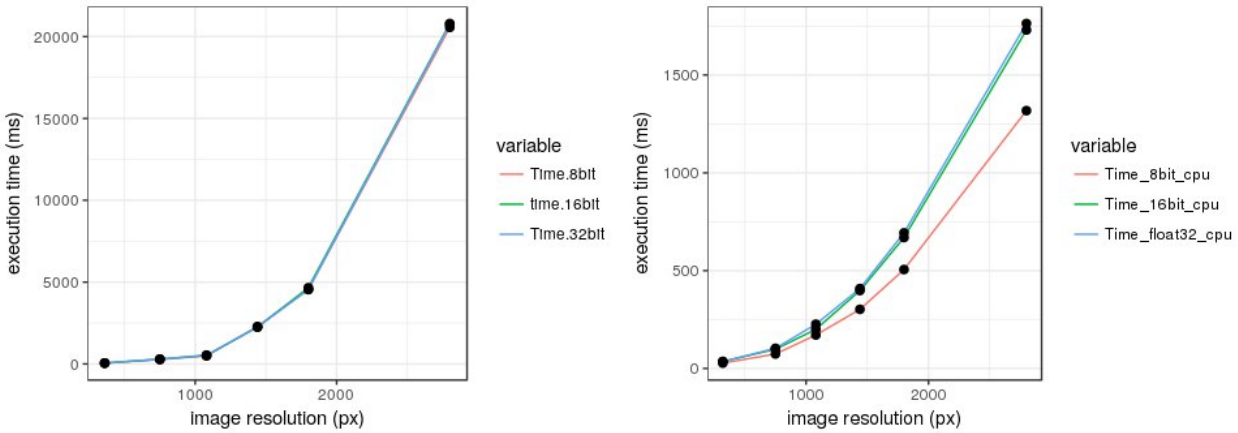

Fig 15. Execution time benchmark analysis with two different methods to compute the variance, one based on integral image (left) against single pass method (right) for a circular kernel of radius = 1 and for 3 different types of image (8bit,16bit and float32).

On the figure 15, the execution time for either 8bit, 16bit or float32 for an image with the same resolution does not change significantly on either resolution, in fact the 3 lines which represent the execution time are close together between the two methods except for the 8 bit filter that is way more faster than for the two other types because of the low complexity values [0...256]. However, the two methods differ by a factor of 1000, that can be explain by the fact that for the integral image's method it iterates many times through the image to compute the variance whereas the one pass only iterate once. Hence, the single pass method fit more to a GPU implementation mainly because of his execution time, way more faster than the integral image method and easier to implement.

Benchmark CPU vs GPU and ImageJ for two different kernel size.

This part is mainly focused on demonstrate the gap between the CPU implementation of the single pass against the GPU and ImageJ implementation of the variance filter .

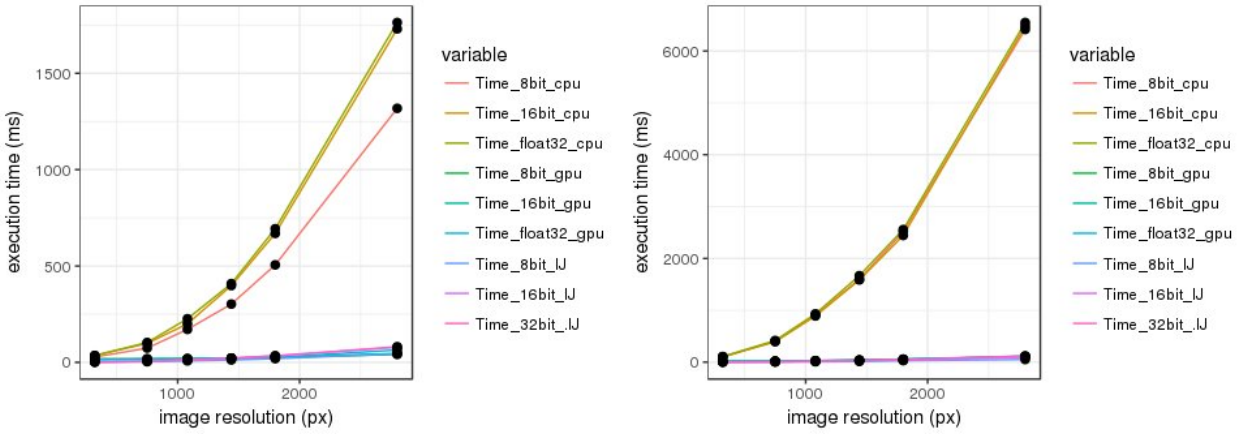

Fig 16. Execution time of the variance filter for three different implementation of the one pass algorithm with ImageJ, CPU and GPU with the same algorithm. The left image represent the execution time of the variance filter for a kernel radius = 1 when the right image represent the execution time of the variance filter for a kernel radius = 3 and for 3 different types of image (8bit,16bit and float32).

On the figure 16, the increase of the execution time for both ImageJ and GPU stays globally the same for the different resolutions while the CPU highly increased. Indeed for a kernel radius the difference between the two differ by a factor of nearly 500/600 for a resolution of 1880x1440 pixels when it's triple for a resolution of 2880x2304 pixels with a factor close to 1500. The same pattern can be seen for a radius equal to 3 except that's the execution time for the cpu is 2000 better for a resolution of 1880x1440 pixels and three times more for the next resolution.

Benchmark GPU vs ImageJ

As we previously described that the GPU and ImageJ implementation are more faster than the CPU in term of execution time, what about the performance between the GPU and ImageJ for two different kernel.

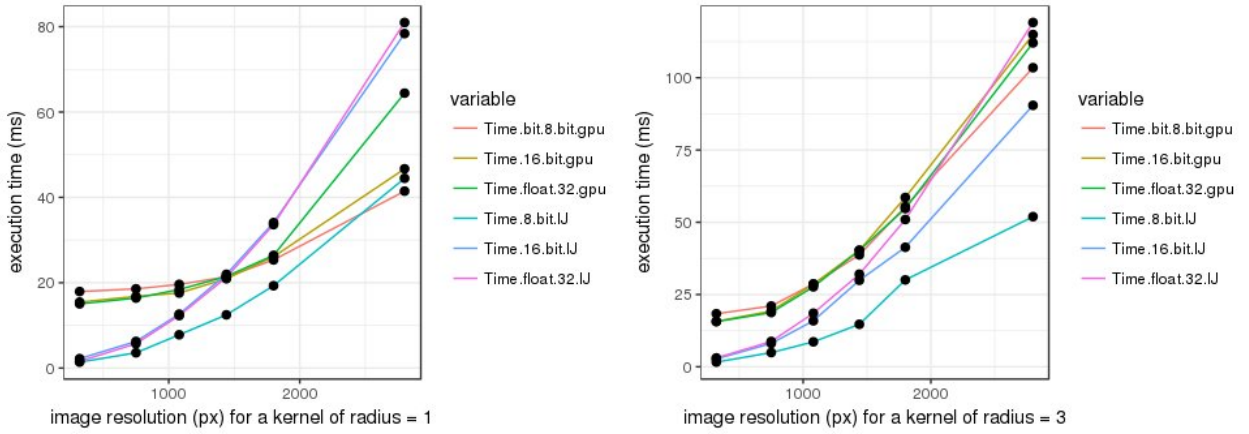

Fig 16. Execution time benchmark analysis of our GPU against the variance algorithm of ImageJ for a kernel radius of 1 for the left part and a kernel radius of 3 for the right part and for 3 different types of image (8bit,16bit and float32).

On the figure 16, the execution time from the first resolution to the fifth doesn't really change, also the scale of the benchmark isn't the same, even if the ImageJ algorithm is more efficient by nearly 10ms than the GPU for the four first resolutions. However, the GPU filter is more stable than the ImageJ filter except for the float 32 who highly increased for a resolution of 1880x1440 pixels. We can note that at the fourth resolution (1080x864) the GPU become more performant than ImageJ for the 16 and 32 bit. Despite the fact that the 8 bit filter is more performant for ImageJ in the early stage, that difference decrease with higher resolutions. Indeed, if they were 20 ms of difference between ImageJ and GPU for the 8 bit filtering at the start, for a resolution of 2880x2304 pixels, ImageJ and the GPU are on equal footing. For bigger kernels we can see the same pattern except that overall it seems that ImageJ create a gap when it comes to higher resolutions with a factor 2 of difference when it reach a resolution of 2880x2304 pixels. Although, ImageJ is performing better for 8 bit images at higher resolution and with bigger kernels the performance stays very similar between those two when comparing the 16 bit and float 32 that are closed to 100 ms for a resolution of 2880x2304 pixels.

4.Discussion

Overall quality comparison between imageJ, CPU and GPU for the variance filter.

The implementation of the naive algorithm in Webgl tends to gain in term of effectiveness comparing to the CPU integral image, whereas the ouput image is the same for the CPU and GPU based on the one pass algorithm. By comparing the result of the image with ImageJ against our GPU, even if the subtracting of those two shows many black pixels compared to the integral image with ImageJ it's not quite still similar. Indeed, ImageJ adjust the brightness and contrast after having computed the variance, while it's not the case for our algorithm. In term of implementation, I discuss with the group of edge detection to find a

solution in order to find a threshold that fit perfectly with the different raster types. The code is present in gpuVariance.js, however it was not successful.

Overall performance comparison between imageJ, CPU and GPU for the variance filter.

The CPU filter for the both implementation (integral image and one pass algorithm) are increasing exponentially and the execution time between those two differ by a factor 1000 in favor of the one pass algorithm. Indeed, the integral image is more complex to implement and iterates over the image to many times that explain that difference. In the same time, the CPU execution time for the one pass algorithm compared to the implementation of ImageJ and GPU differs by a factor between 500/600. This can be explained by the fact that there's no multi-threading that treat each pixels in parallel.

In terms of performance, the GPU implementation and ImageJ implementation are behaving similarly. For a kernel of 3x3 the GPU, is not as fast for small resolution between 360x288 and 900x72. However when comparing with bigger resolution the GPU implementation act faster than Image implementation by around 20ms except for the 8bit that's the same. When comparing those two implementation with a kernel of 7x7, the same pattern is observed ImageJ is faster than the GPU at start and that difference decrease for bigger resolution and this time the 8bit implementation of ImageJ is around 20ms faster than the GPU.

When I try to add the step implementation for the thresholding the execution time was between 2 and 3 times longer than without it. Hence, it was not cost effective to implement it.

5.Conclusion

Results obtained with our filter are quite similar in term of execution time with those observed in ImageJ for 8,16 and float32. However there is some slightly difference between the both output image, mainly because ImageJ adjust the brightness and contrast . The execution time of our GPU implementation is much faster than the CPU implementation because of the parallelization of the treatment of each pixel, which mean our implementation has been successful. Further improvement can be added for the variance filter webgl implementation, first will be to use a texture that fit more for the 16 bit. Moreover, it's very hard to find the perfect threshold value for the variance filter as so, it will be interesting to add some widgets in webgl to manually change the threshold in function of the image resolution or the kernel size. The result may have been similar if our implementation add a feature that adjust the brightness and contrast. Finally, an implementation of the integral image may be useful to not have to find an arbitrary threshold for the variance filter. As conclusion, this project has been very instructive mainly for the different knowledge assimilate (benchmark, js, webgl, bibliography research, team work, adaptation to the library TIMES)