

wavelet tranform

Advanced Computer Architecture – A.A 2019/2020

[Data]

University of pavia

Giuseppe Carniglia, Roberto Nour

**INDEX**

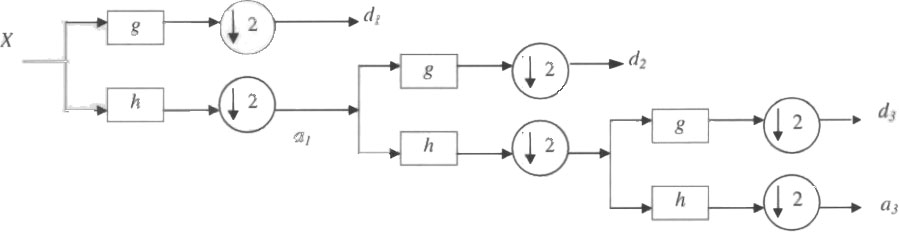
* Introduction………………………………………………………………………………………… p.1
* Haar wavelet transform and 9/7 wavelet transform……………………………. p.1
* Haar wavelet transform………………………………………………………………………. p.4
  + 1.General algorithm………………………………………………………………….. p.4
  + 2.Results……………………………………………………………………………………. p.5
  + 3.Antitrasformation…………………………………………………………………… p.5
  + 4.Antitrasformation results……………………………………………………….. p.6
* 9/7 wavelet transform…………………………………………………………………………. p.6
  + 1.General algorithm…………………………………………………………………… p.6
  + 2.First step - problem of empty pixels……………………………………….. p.7
  + 3.Central pixels………………………………………………………………………….. p.8
  + 4.Possible errors with images…………………………………………………….. p.8
  + 5.Results…………………………………………………………………………………….. p.8
* Parallelism……………………………………………………………………………………………. p.9
* Open points about the overflow…………………………………………………………… p.10
* Computation on GCP……………………………………………………………………………. p.12

**INTRODUCTION**

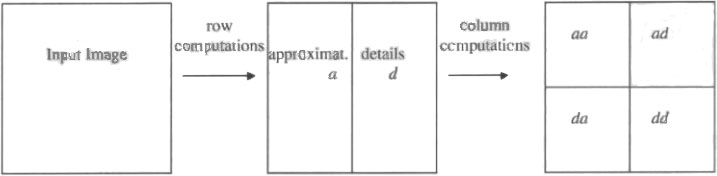
When we look at images, generally we see connected regions of similar texture and intensity levels that combined form shapes. If the shapes are small or low in contrast, we normally examine them at high resolutions; if they are large or high in contrast, a coarse view is all that is required. If both small and large shapes (or low and high contrast objects) are present simultaneously, it can be advantageous to study them at several resolutions. This, of course, is the fundamental motivation for multiresolution processing. From a mathematical viewpoint images are two-dimensional arrays of intensity values with locally varying statistics that result from different combinations of abrupt features like edges and contrasting homogeneous regions.  
The discrete wavelet transform (DWT) has allowed to consider many ideas and techniques, commonly used in different fields of application. Wavelets are a versatile tool to represent efficiently functions or sequences of data. Their main properties are the space-frequency localization, their inherently multiresolution structure and, from an application point of view (discrete case), the existence of fast computational algorithms with linear complexity. Wavelets allow an efficient representation over various scales, they allow to retain the information in a signal, providing a compact representation, taking advantage of the data correlation in space and frequency.

**HAAR WAVELET TRANSFORM AND 9/7 WAVELET TRANSFORM**

The digital wavelet transform is usually computed through convolution and sub-sampling with a couple of filters to produce an approximation signal a (low pass filter result) and a detail signal d (high pass filter result). The multiresolution decomposition is obtained by iterating the convolution and sub-sampling of these two filters over the approximation components. For 2-D signals there exist separable wavelets, for which the computation can be decomposed into horizontal processing (on the rows) followed by vertical processing (on the columns), using the same 1\_D filters. The other levels are obtained iterating on the low pass signal aa.



A 3 level 1-D wavelet decomposition. The multiresolution representation is obtained by iteration on the approximation signal a.



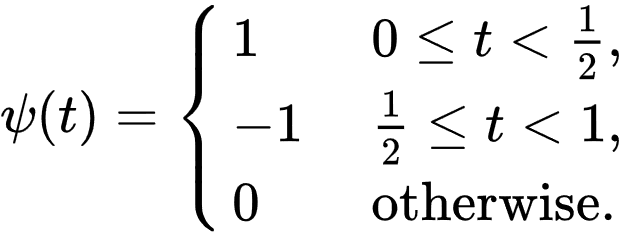
A 1-level 2-D separable wavelet decomposition. The next iteration is on the signal aa only.

**HAAR WAVELET TRANSFORM**

The first DWT was invented by the Hungarian mathematician Alfréd Haar.

In mathematics, the Haar wavelet is a sequence of rescaled "square-shaped" functions which together form a [wavelet](https://en.wikipedia.org/wiki/Wavelet" \o "Wavelet) family or basis. Wavelet analysis is similar to [Fourier analysis](https://en.wikipedia.org/wiki/Fourier_analysis" \o "Fourier analysis) in that it allows a target function over an interval to be represented in terms of an [orthonormal basis](https://en.wikipedia.org/wiki/Orthonormal_basis" \o "Orthonormal basis). The Haar wavelet is also the simplest possible wavelet. It’s technical disadvantage is that it is not [continuous](https://en.wikipedia.org/wiki/Continuous_function" \o "Continuous function), and therefore not [differentiable](https://en.wikipedia.org/wiki/Derivative" \o "Derivative). This property can, however, become an advantage for the analysis of signals with sudden transitions such as monitoring of tool failure in machines.

The Haar wavelet's mother wavelet function ψ(t) ψ ( t ) {\displaystyle \psi (t)} can be described as



So the Haar transformation can be expressed in the following matrix form:

***T = HF***

Where F is an N x N image matrix, H is an N x N Haar Transformation matrix, and T is the resulting N x N transform(The transpose is required because H is not symmetric)

-GENERAL ALGORITHM

For an input represented by a list of 2n numbers, the Haar wavelet transform may be considered to simply pair up input values, storing the difference and passing the sum. This process is repeated recursively, pairing up the sums to provide the next scale: finally resulting in 2n-1 differences and one final sum.

Suppose you are given N values:

sk = (x2k + x2k+1)/2 for k=0, …, N/2 -1

For example,

x = (6, 12, 15, 15, 14, 12, 120, 116) -> s = (9, 15, 13, 118)

We need second list of data **d** so that the original list x can be recovered from **s** and **d**.  
For dk (called directed distances), we have:

dk = (x2k - x2k+1)/2 for k=0, …, N/2 -1

The process is invertible since:

sk + dk = (x2k + x2k+1)/2 + (x2k - x2k+1)/2 = x2k

sk - dk = (x2k + x2k+1)/2 - (x2k - x2k+1)/2 = x2k+1

So we map **x** = (x1, x2, … , xN) to (**s | d**) = (s1, … , sN/2 | d1, … , dN/2).

Using our example values, we have:

(6, 12, 15, 15, 14, 12, 120, 116) -> (**9, 15, 13, 118** | -3, 0, 1, 2)

This process is repeated recursively for **s**:

(**9, 15, 13, 118** | -3, 0, 1, 2) -> (**12, 65.5** | -3, -52.5 | -3, 0, 1, 2)

(**12, 65.5** | -3, -52.5 | -3, 0, 1, 2) -> (**38.75** | -26.75 | -3, -52.5 | -3, 0, 1,2)

So final result is:

(38.75, -26.75, -3, -52.5, -3, 0, 1, 2)

Why might people prefer the data in this form?

* We can identify large changes in the differences portion **d** of the transform.
* It is easier to quantize the data in this form.
* The transform concentrates the information (energy) in the signal in fewer values

In case of images, we need 2D FWT. First, we perform 1D FWT for all rows, and next, for all columns.

1. RESULTS



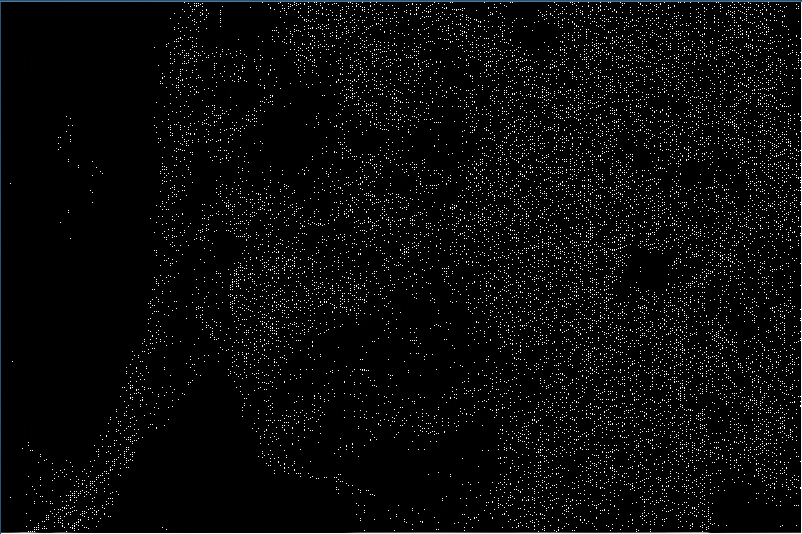
- ANTITRASFORMATION

To see if our algorithm works properly, we computed the Haar transformation, of which we know the formula:

x2k = sk + dk = (x2k + x2k+1)/2 + (x2k - x2k+1)/2

x2k+1 = sk - dk = (x2k + x2k+1)/2 - (x2k - x2k+1)/2

With this operation we see that our algorithm seems to work properly, because the image that we obtain with this transformation is very similar to the original image; also subtracting pixel by pixel the two images reveal also that this DWT is lossy, since the resulting image is mostly black (the subtracted pixel values where equal) but there are some white/light gray pixels confirming that it is a lossy transformation.



1. ANTITRASFORMATION - RESULTS



**9/7 WAVELET TRANSFORM**

* GENERAL ALGORITHM

The structure of our algorithm is the same as for the Haar but this time we have, as the name implies, nine coefficients for the low-pass filter and seven for the high-pass. As mentioned above the low-pass filter is used to scale down the original image and the high-pass filter to select the contours of the shapes in the image.

Below are reported the values of these coefficients:



The iterative algorithm operates firstly on the rows and secondly on the columns.

Every pixel of the transformed image is obtained as a discrete convolution among the corresponding pixel of the original image and the 8 pixels surrounding it (4 on the left and 4 on the right) convoluted with the filter coefficients. This computation is performed having as central pixel only the even ones to achieve the subsampling by two expressed in the above diagram.

2.FIRST STEP – PROBLEM OF EMPTY PIXELS   
Since the first operation that we do is on the rows, we kept constant its index in the loop and modified the column index. The first 4 left pixels of each row don’t have 4 pixels to their left, since the image does not extend further in that direction, so to complete the calculation we padded the image with 4 columns and rows of zeros.

In fact, here we find the macro “ZERO\_VALUE” and the same consideration is for the last 4 pixels on the right because in this zone there aren’t enough pixels to the right.

|  |  |  |
| --- | --- | --- |
|  |  | X |

|  |  |  |
| --- | --- | --- |
| X | X | X |

|  |  |  |
| --- | --- | --- |
|  | X | X |

X X X

This is a graphical representation of the first 3 pixels to explain what we mean when we say that we haven’t some pixel on the right (in this case) and on the left is just specular. The black cross is the middle pixel (the one multiplied by the coefficients in position zero), the red cross is the pixel needed for the computation but that is non existing because outside of the image (it’s value is assumed to be zero with the macro ZERO\_VALUE) and the green pixels are the ones involved in successive calculations that are now in the image range and can be used to compute a single pixel of the transformed image.

Immagine che contiene tavolo, sedendo, dilegno, telefono

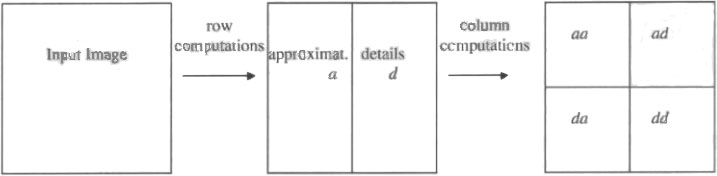
Descrizione generata automaticamenteCode in which we made all these operations.

3. CENTRAL PIXELS

For all pixels in the center we compute them using all the relative values. With this procedure each sum of products gives us the new pixel of the modified image. Once the operations for the rows are completed using low-pass coefficients the same operation is repeated using high-pass ones.

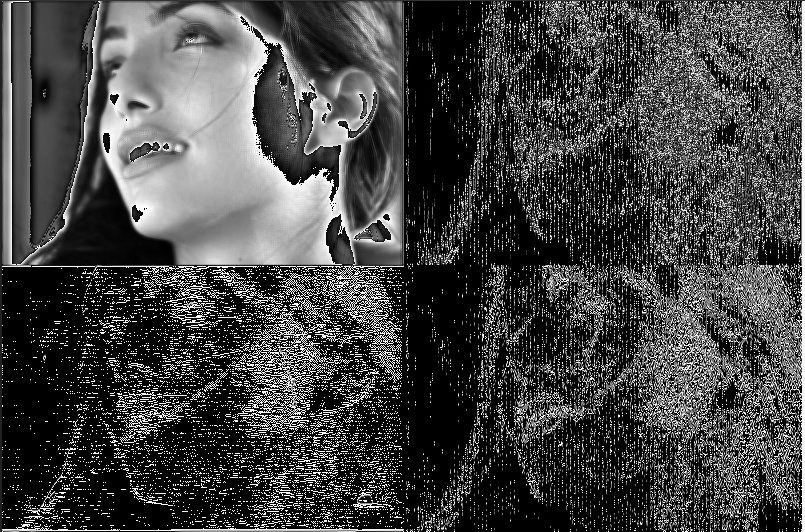
The result is an image with same dimension as the original, on the left the result of low-pass filtering is the original image halved in width, on the right the result of high-pass filtering are the details of the original image.

The second phase consists in applying a similar procedure on columns using the previously obtained image as input. This time for the top half of the image low-pass coefficients will be used and for the lower part the high-pass ones creating four images: on top left a copy of the original scaled down to a ¼ of the original dimension(aa), on the bottom right the image of details(dd) reduced in size to half the previous height, on top right and bottom left are images that where influenced by both low-pass and high-pass filtering (ad, da)



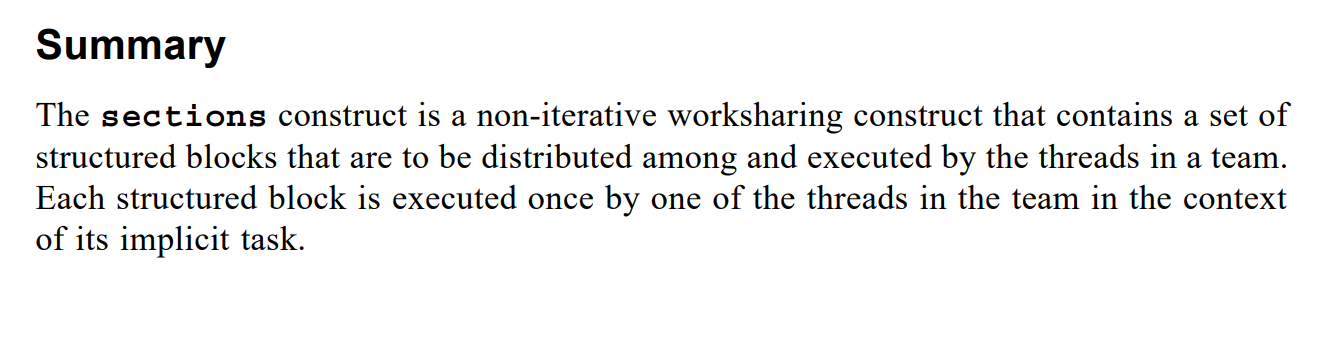
-POSSIBLE ERRORS WITH IMAGES.  
We used temporary images to avoid overriding the previously modified image and avoid inconsistent results or errors due to erroneous pixel values. At the end of calculations, the temporary images are put together in a final image.

5. RESULTS

Here we can see the four images after the computation of the algorithm: 

**PARALLELISM**

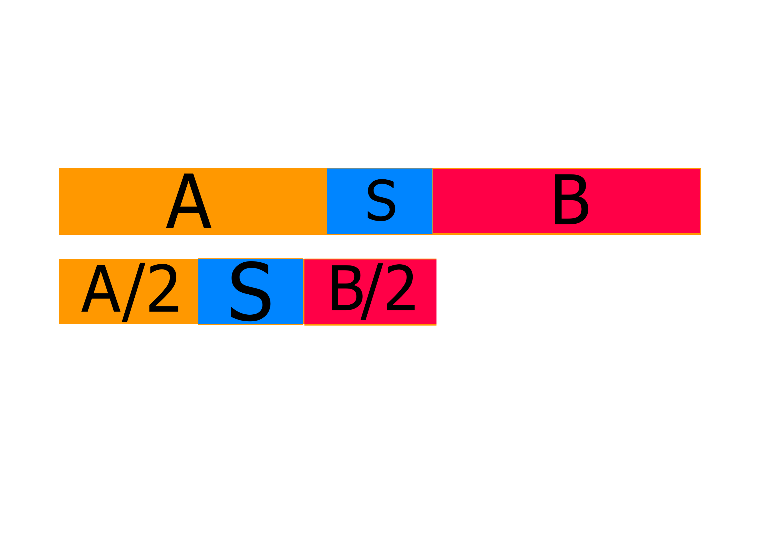
While studying the algorithm we understood that the only necessary serial part was the succession of row calculations and column calculations, since the second needs the result of the first to proceed. So, everything else could have been parallelized.

Our first attempt consisted into dividing the job of computing each image (a,d or aa,da,ad,dd) using sections and assigning some threads to each section to speed up the computation. We implemented this behavior by creating sections in a parallel region; each section called a function, passing proper arguments, to compute the row or column calculations. Inside these functions we also added the directives to parallelize a for cycle but noticed the fact that having a parallel cycle in a section didn’t create new threads, but instead used only the thread assigned to that section. So the threads were generated in the parallel region and then assigned to the sections and all code inside the section was managed by a single thread. 

[OpenMP Application Program Interface Version 4.0 - July 2013 pag.60]

We thought about another solution. Since the computations on low-pass and high-pass values are independent we decided to merge the two for loops in each phase together to carry out the computations. Also, since the original image is always read but never wrote concurrently the two loops allow us to group the total computation for the first phase into a single loop and subsequently using the proper directives parallelizing it allocating the maximum number of threads possible on the machine to execute the calculation. The second phase being so similar to the first has the same solution with the only exception that temporary images are used to ensure that concurrent write operation could not happen. We believe is not a detrimental solution since at maximum four such images are used for each iteration of the algorithm and their size reduces exponentially with the number of the iteration, as such the memory consumption should not be a major concern.

**Amdhal’s Law**

In both our programs the structure is very similar, two parts of computation for possible parallelization and the middle part serial to move from one computation to the other. 

Since two parts can benefit from the parallelization in an optimal case, we expect a speed-up for both parts on two threads equal to two on the overall speed-up of the programs. But since there is the small serial part the total speed-up is less.

For the Haar serial transform execution times divided in total, first calculation, serial, second calculation are (in seconds):

 Haar serial: 0.782613, 0.529001, 0.000002,0.253610

As we can see the serial part in this case is 2,55\*10^-6 so is practically negligible, the part that benefits from the speedup is (0.529001+0.253610)/ 0.782613 = 0.999 so the 99,9%

Hence the theorical speed-up according to the Amdahl law is (2 threads):

S = , since P = 0.999 and s=2 we obtain S = 1,998

This being a theorical computation is just an indication on how much of a speedup to expect, in reality after our parallelization the real speedup is:

Haar parallel: 0.742667,0.376502,0.00019,0.366146

S(%) = (1 – (0.742667 / 0.782613)) \*100 = 5.10%

Which is almost and entire point lower than the theoretical expected value. We think that the reasons behind this difference are due to overheads due to OpenMP and maybe a too simple parallelization strategy. But thanks to GCP we were able to test the programs on more than one cpu and found out that the speedup increases at higher CPU count reaching at 16vCPU

S4(%) = (1 – 0.287937/0.334300) \*100 = 13.86% (Amdahl S4 = 3.988)

S8(%) = (1 – 0.21463972/0.3671365) \*100 = 13.86% (Amdahl S8 = 7.944)

S16(%) = (1 – 0.18083864/0.31064279) \* 100 = 41.7% (Amdahl S16 = 15,76)

For the CDF 9/7 a similar approach has been considered hence only the theoretical and practical results are shown.

CDF 9/7 serial: 1.58700,0.796128, 0.005928, 0.784944

P = (0.796128 + 0.784944)/ 1.58700 = 0,996

S = , P= 0,996 s=2 so S = 1,992 Amdahl speedup

CDF 9/7 parallel: 0.788352, 0.395193, 0.000001, 0.393158

S(%) = (1 - 0.788352/1.58700) \* 100 = 50.32%

And with GCP vCPUs(4,8 and 16 vCPUs):

S4(%) = (1 - 0.428150 /0.704544) \* 100 = 39.23% (Amdahl S4 = 3.95)

S8(%) = (1 - 0,20815271/0,62249513) \* 100 = 66.56% (Amdahl S8 = 7.78)

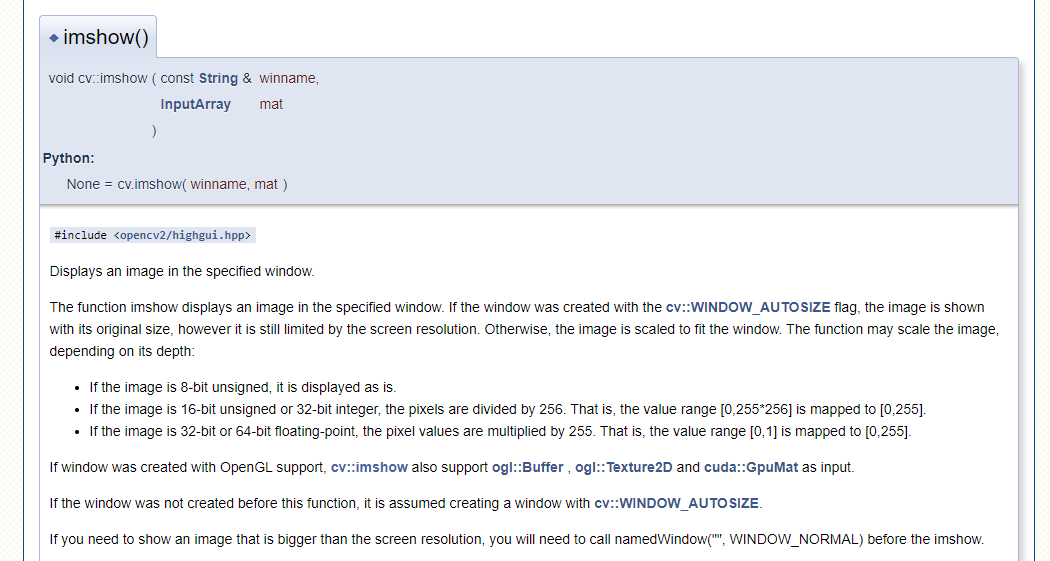
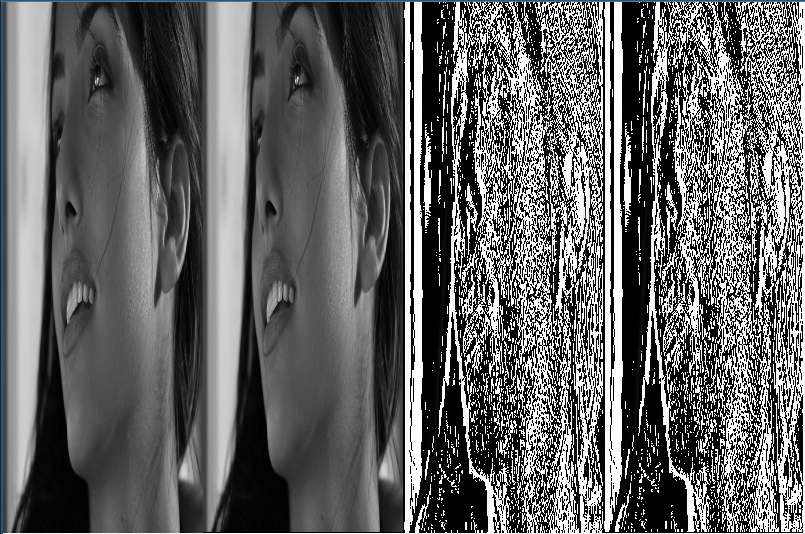
S16(%) = (1 - 0,10579385/0,65515247) \* 100 = 83.85% (Amdahl S16 = 15,09)

**OPEN POINTS ABOUT THE OVERFLOW:**

First of all we tried to normalize the values of each single pixel of the image at the end of all operations. The problem of overflow rises from the fact that each pixel in a gray scale is encoded with 8 bit unsigned UCHAR (U-> unsigned and Char-> character with 8 bits size).

This problem showed up in a visible way only with the 9/7 algorithm and not in the computation of Haar because in this last one we have simpler computations. In the 9/7 we have a more complex computation since we have sums of multiplications. Here we have all coefficients that are less than 1 (this is for both low pass coefficient and high pass coefficient) but having all these sums at the end we overflow on 8 bit.

To solve this we created the matrix that would have contained the image with bigger types to accommodate overflowing values on 8 bit, the types of choice were USHORT and INT.

With the USHORT solution the problem of the range was solved, but we encounter another problem due to how OpenCV manages the values higher than 8 bits.   
Our thought for this problem is about the fact that we notice that 16 bit is exactly the double of 8 bit and when we visualized the image, on the left were two vertical lines with two times the copy of image in black and white because OpenCV with a size of 16 bits copied the 8 bits of the image in the superior 8 bit of the 16, so it filled the 16 bit with a copy of 8 bit of the image. At the end the total image was two times the original image, so it duplicated the image.  
We try also to use the INT values, but the problem just described is increased because with INT value we have 32 bits that is 4 times 8 and what we see was 4 vertical lines with the copy of image.

**COMPUTATION ON GCP.**

To test our implementation of parallelism for both wavelets and to see if there is an actual improvement in execution time we created 4 virtual machines on Google Cloud Platform and for each program executed two identical test: first test was to compute the transform of one image (immagini/leonessa.jpg, 3840x2160) 100 times to check the average time of execution, the second was to compute the transform on 6 different images with same resolution in an attempt to force cache’s swap. In the end the average times of execution for each program are plotted over the number of VM’s CPUs.

As it’s possible to see from the picture, the average time of execution for parallel Haar tends to decrease almost linearly while the serial presents a higher value at 8vCPU and decreases at 16vCPU, we cannot explain this behavior.

While the 9/7CDF serial transform presents the lowest average time at 8vCPU, the parallel presents an almost linear descending trend. We cannot explain the sudden increase in time at 16 vCPU with respect to 8vCPU.

The following pictures describe in detail for each program at each vCPUs configuration the average execution time and execution times at every attempt