

Università di Pisa and Scuola Superiore Sant'Anna



Master Degree in Computer Science and Networking

Master Thesis

## **Parallelizing Image Processing Algorithms For Face Recognition on Multicore Platforms**

**Candidate:** Valdo João

**Supervisors:**

Prof. Giorgio Buttazzo

Prof. Marco Danelutto

**Referee**

Dott. Antonio Cisternino

Academic Year 2016/17

## Table of Contents

<b>1. Introduction .....</b>	<b>3</b>
1.1 Challenges associated with face detection .....	4
1.2 Closed related problems of face detection .....	5
1.3 Face detection methods .....	6
<b>2. Real-Time Systems .....</b>	<b>8</b>
2.1 Real-Time Face Detection.....	8
<b>3. Viola-Jones Face Detection Framework.....</b>	<b>11</b>
3.1 Integral Image .....	11
3.2 AdaBoost learning algorithm .....	13
3.3 The attentional Cascade .....	14
3.4 Training a Cascade of Classifiers.....	16
3.5 Failure modes of Viola-Jones algorithm .....	17
<b>4. Implementation of Viola-Jones Algorithm with OpenCV .....</b>	<b>18</b>
4.1 Face Detection process.....	20
4.2 Face Tracking.....	24
4.2.1 Evaluation of Existing Object Tracking Algorithms .....	25
4.2.2 Considerations about the object tracking algorithms .....	27
<b>5. Sequential Implementation .....</b>	<b>29</b>
5.1 Optimizations .....	30
5.1.1 Face detection optimization .....	30
5.1.2 Feature detection optimization .....	35
<b>6. Parallel Implementation.....</b>	<b>37</b>
6.1 Parallel Patterns and Tools.....	37
6.2 Parallel implementation of OpenCV .....	40
6.3 Sources of Parallelism.....	41
6.4 Additional Parallelism to the OpenCV Viola-Jones Implementation .....	46
6.4.1 General Approach: Multiple Face Detection.....	47
6.4.2 Specific Approach: Single Face Detection .....	52
6.4.3 Feature Detection Parallelization.....	53
6.5 Parallel Implementation with GPUs.....	54
6.5.1 GPU Parallelization with CUDA.....	54

6.6 Unbalanced Computation of Viola-Jones on Multi-Core Platforms .....	57
6.6.1 The GPU Case .....	58
6.6.2 The CPU Case .....	61
<b>7. Tests .....</b>	<b>64</b>
7.1 Pipeline stages .....	65
7.2 Multiple and single Face Detection.....	66
7.2.1 Tests for the general case: Multiple Face Detection.....	66
7.2.2 Tests for the specific case: Single Face Detection .....	68
7.3 Real-Time Coverability on Video Frames .....	70
7.4 Number of Sub-Windows Processed by the Stages of Cascade Classifiers.....	74
7.5 Reliability Tests.....	75
7.6 Coexisting Different Parallel Programming Frameworks.....	77
7.7 Summary of the Results Achieved .....	78
<b>8. Conclusions.....</b>	<b>80</b>
<b>9. References.....</b>	<b>82</b>

## 1. Introduction

Humans have a great ability to analyze images. They can detect (recognize) faces in an extremely reliable way. Indeed, humans are able to easily locate faces in its environment despite difficult conditions such as occlusions of parts of a face and bad lightening.

Many studies have been conducted to try to replicate this process automatically, using machines, because face detection is considered as a pre-requisite for many Computer Vision application areas such as security, surveillance and content based image retrieval.

Computer vision is a field devoted to provide high-level understanding of images. It aims to determine what is happening in front of a camera and use that understanding to control a computer or robotic system, or to provide new images that are more informative than the original ones. In order to build fully automated systems that analyze the information contained in face images, robust and efficient face detection algorithms are required.

The human face is a dynamic object and has a high degree of variability in its appearance which makes face detection a not so easy problem to deal with [1]. One of the most important and necessary conditions for face detection is the exclusion of the background for reliable face classification techniques.

However, it is possible that a face could occur in a complex background (Fig. 1) and in many different positions, in this case face detection systems are likely to mistake some areas of the background as a face [1].

The solution for the face detection problem involves segmentation, extraction and verification of face and possible facial features from the background. A face detection system should be able to achieve this task regardless of illumination, orientation and camera distance.

However, in practice it is not possible to build a simple and rapid detector with high precision and response to all the possible face images variations because of the big interclass variance, the variety of ambient light conditions, as well as the complex structure of the background [10].

The face detection problem can be defined as: given an image or video sequence, determine whether or not there are any faces in the image and if present, return the image location and extent of each face [2].

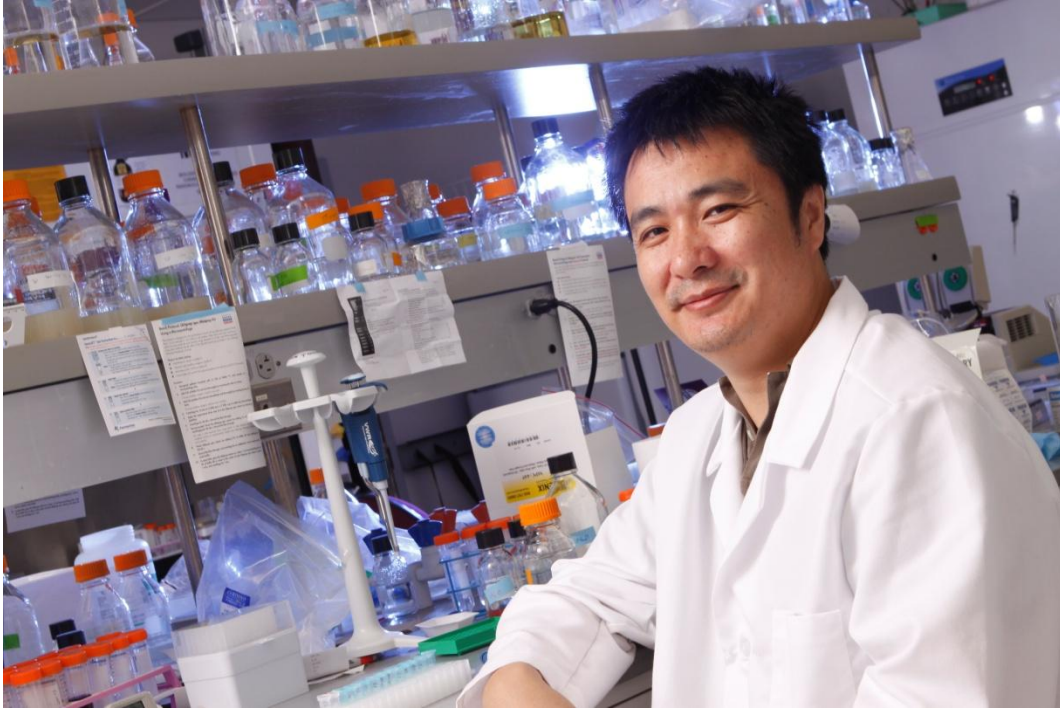


Figure 1: A REALISTIC FACE DETECTION SCENARIO.

## 1.1 Challenges associated with face detection

The challenges associated with face detection can be attributed to several factors. They can be named differently from author to author but the problems addressed are the same.

- a) **Pose:** the images of a face vary due to the relative camera-face pose (frontal, 45 degree, profile, upside down) and some facial features such as an eye or the nose may become partially or wholly occluded.
- b) **Presence or absence of structural components:** facial features such as beards, mustaches and glasses may or may not be present and there is a great deal of variability among these components including shape, color and size.
- c) **Facial expression:** the appearance of face is directly affected by his facial expression.
- d) **Occlusion:** faces may be partially occluded by other objects, for instance in an image with a group of people, some faces may partially occlude others.
- e) **Imaging conditions:** factors such as lighting (spectra, source distribution and intensity) and camera characteristics (sensor response, lenses) affect the appearance of a face.

## 1.2 Closed related problems of face detection

There are many closed related problems of face detection. The most common ones are:

- a) **Face Localization:** aims to determine the image position of each single face.
- b) **Facial feature detection:** aims to detect the presence and location of features such as eyes, nose, mouth, lips, etc.
- c) **Face identification:** compares an input image against a database and reports a match, if any.
- d) **Face authentication:** verifies the claim of the identity of an individual in an input image.
- e) **Face tracking:** estimate the location and possible orientation of a face in an image sequence.
- f) **Facial expression recognition:** concerns identifying the affective states (happy, sad, disgusted, etc) of humans.

Evidently, face detection is the first step in any automated system which solves the above problems.

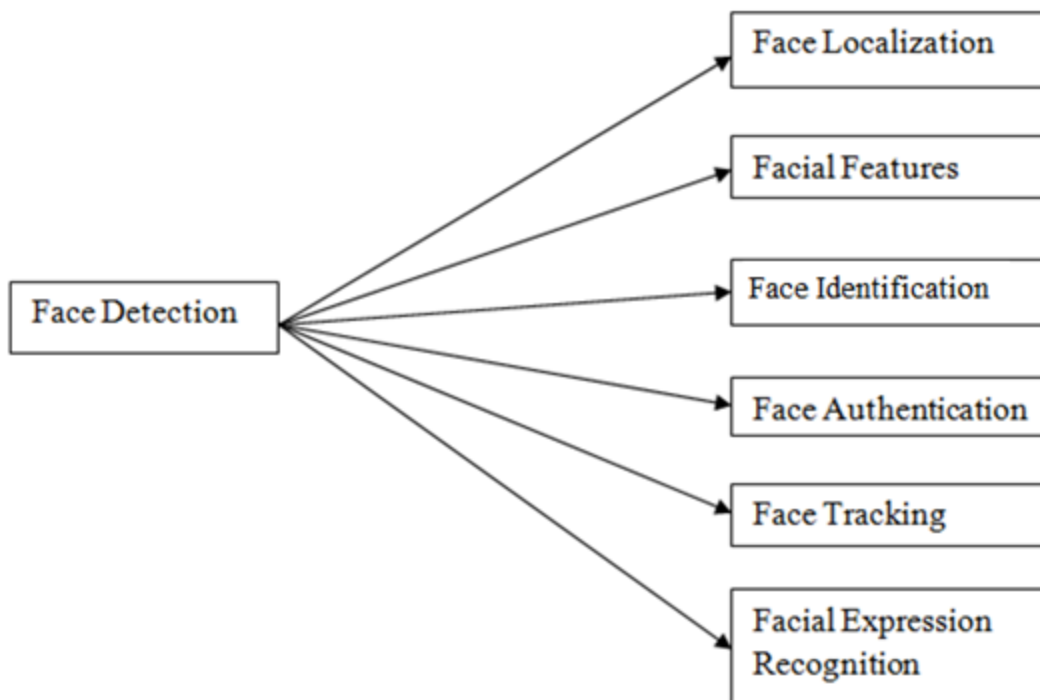


Figure 2: CLOSED RELATED PROBLEMS OF FACE DETECTION

Before going forward, it's important to define two basic concepts that refer to the types of errors that face detectors can make.

- **False negatives:** are the cases in which faces are missed resulting in low detection rates.
- **False positives:** are the cases in which an image region is declared to be a face but actually it is not.

### 1.3 Face detection methods

Several authors have defined a vast range of methods for face detection but all of them fall into one of the categories described by [2].

**Knowledge-based methods:** this technique encodes human knowledge of what constitutes a typical face. In this approach, face detection methods are developed based on the rules derived from the researcher knowledge of human faces. It is easy to come up with simple rules to describe the features of a face and their relationship. For example, a face often appears in an image with two eyes that are symmetric to each other, a nose and a mouth.

One problem with this approach is the difficulty in translating human knowledge into well-defined rules. If rules are detailed (strict), they may fail to detect faces that do not pass all the rules. If the rules are too general, they may give many false positives.

**Feature invariant approaches:** aims to find structural features that exist even when the pose, viewpoint or lighting conditions vary. The assumption is that humans can effortlessly detect faces and objects in different poses and lighting conditions, so there must exist properties or features which are invariant over these variabilities.

One problem of this approach is that the image features can be severely corrupted due to illumination, noise and occlusion.

**Template matching method:** defines several standard patterns of a face which are stored on a template in order to describe the face as a whole or the facial features separately. The correlation between input and stored patterns are computed for detection.

This approach has the advantage of being simple to implement. However it has proven to be inadequate for face detection since it cannot effectively deal with variation in scale, pose and shape.

**Appearance based methods:** in contrast to template matching, the models (or templates) are learned from a set of training images which should capture the representative variability of facial appearance.

In general, this approach rely on techniques from statistical analysis and machine learning to find the relevant characteristics of face and non-face. One problem of this approach is that it can take hours, even days to train the models.

### **Considerations about face detection methods**

The boundary between knowledge-based methods and template matching methods is blurry since the latter usually implicitly requires human knowledge to define the face templates. Among the face detection methods, the ones based on learning algorithms (appearance based) have attracted much attention recently due to the fact that eliminates the potential modeling error as a result of an incomplete or inaccurate face knowledge.



## 2. Real-Time Systems

Real-Time Systems are computing systems that must perform computation within given timing constraints. It is a system in which the correctness depends not only on the output values, but also on the time at which results are produced [3]. A correct action executed too late can be useless or even dangerous.

The goal of a real-time system is to guarantee the timing behavior of each individual task. When modeling and analyzing a real-time application is important to predict worst-case response times and verify its feasibility under a set of constraints. Predictability of response times must be guaranteed for each critical activity and for all possible combination of events.

The operating system is the most important component responsible for achieving a predictable execution [3]. A real-time operating system is responsible for:

- Managing concurrency;
- Activating periodic tasks at the beginning of each period (time management);
- Deciding the execution order of tasks (scheduling);
- Solving possible timing conflicts during the access of shared resources (mutual exclusion);
- Manage the timely execution of asynchronous events (interrupt handling).

A real-time task can be distinguished in three categories:

- Hard: a task missing its deadline may cause catastrophic consequences on the whole system.
- Firm: a task missing its deadline is useless for the system, but does not cause any damage.
- Soft: a task missing its deadline has still some utility for the system, although causing performance degradation.

### 2.1 Real-Time Face Detection

Many computer-vision scenarios must be executed in real-time, which imply that the processing of a single frame should be completed within 30 – 50 milliseconds or quasi-real-time 51 - 100 milliseconds [26]. In other words, if we consider frame rates per second (FPS), a real-time scenario is a rate starting from 20 FPS and beyond, a quasi-real-time from 10 FPS to 19 FPS.

The problem of real-time face detection falls under the category of firm real-time tasks in which skipping a video frame is less critical than processing it with a long delay.

From late 90's, early 00's a number of promising face detection algorithms have been developed and published with the aim of achieving real-time face detection . Among all the face detection algorithms two are commonly perceived as the state-of-the-art.

### Neural Network-Based Face Detection [5]

In this face detection framework proposed by Rowley et al in 1998, a rationally connected neural network examines small windows of an image and decides whether each window contains a face. The system arbitrates between multiple networks to improve performance over a single network.

The system operates in two stages: It first applies a set of neural network based filters to an image and then uses an arbitrator to combine the outputs. The filters examine each location in the image at several scales, looking for locations that might contain a face. The arbitrator then merges detections from individual filters and eliminates overlapping detections.

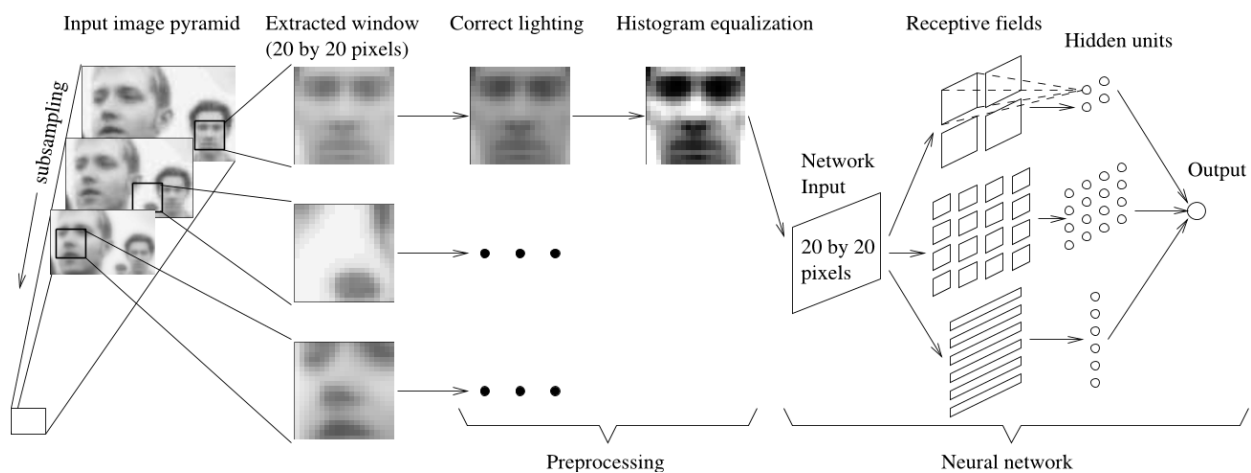


Figure 3: NEURAL NETWORK FACE DETECTION

### Viola-Jones Robust Real-Time Face Detection [6]

This face detection framework was proposed in 2001 by Viola-Jones. At that moment was the first algorithm to provide detection rates in real-time. This face detection framework is capable of processing images extremely rapidly while achieving high detection rates. In order to do so three innovative contributions were added.

The first was the introduction of a new image illustration called the Integral Image<sup>ll</sup> which allows the features to be computed very quickly. The second was an efficient classifier built using AdaBoost learning algorithm to select a small number of critical visual features from a very large set of potential features.

The third contribution was a process for combining classifiers in a cascade which allows background regions of the image to be quickly discarded while spending more computation on promising face-like regions. At that time, this algorithm was 15 times faster than the implementation proposed by Rowley et al.

### **Considerations about the Face Detection Algorithms**

During the years several authors tried to improve both algorithms. Although some improvement was made, the core of the Viola-Jones remains the same while the Neural Network evolved to the so called Deep Convolutional Networks [4].

Several papers tried to discuss which of the approach is the current state-of-the-art pointing out the advantages of one with respect to the other but the difference is blurry. A factor that clearly is in favor of the Viola-Jones algorithm is its solidness as the original paper still the main source of reference.

### 3. Viola-Jones Face Detection Framework

The Viola and Jones face detection framework makes use of three main components in order to process images very fast while achieving high detection rates.

#### 3.1 Integral Image

The first step of the Viola-Jones face detection algorithm is to turn the input image into an integral image. The integral image at location  $x, y$  contains the sum of the pixels above and to the left of  $x, y$  inclusive.

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y'),$$

Using the following pair of recurrences:

- (1)  $s(x, y) = s(x, y - 1) + i(x, y)$
- (2)  $ii(x, y) = ii(x - 1, y) + s(x, y)$

(where  $s(x, y)$  is the cumulative row sum,  $s(x, -1) = 0$ , and  $ii(-1, y) = 0$ ) the integral image can be computed in one pass over the original image.

1	1	1
1	1	1
1	1	1

Input image

1	2	3
2	4	6
3	6	9

Integral image

Figure 4: CONVERSION INPUT IMAGE TO INTEGRAL IMAGE

To achieve a fast and robust implementation, Viola-Jones faces detection algorithm uses rectangle Haar-like features, which are digital image characteristics used in object recognition. There are three types of rectangle Haar-like features (see Fig. 5): two-rectangle features (A and B), three-rectangle features (C) and four rectangle features (D). They operate on grayscale images and their decisions depend on the threshold difference between the sum of the luminance of the white region(s) and the sum of the luminance of the black region(s).

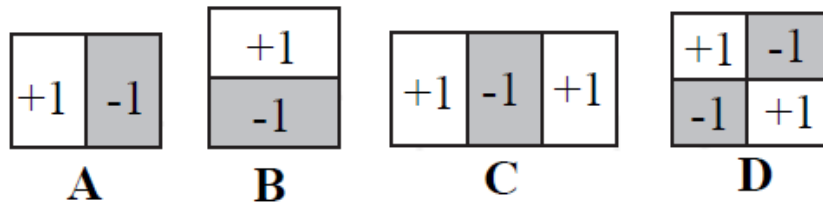


Figure 5: RECTANGLE FEATURES

The integral image allows computing Haar-like features very rapidly at many scales. The calculation of the sum of all pixels inside any given rectangle is made using only four values. These values are the pixels in the integral image that coincide with the corners of the rectangle in the input image. In this way the sum of pixels within rectangles of arbitrary size can be calculated in constant time. This is demonstrated in Fig. 6.

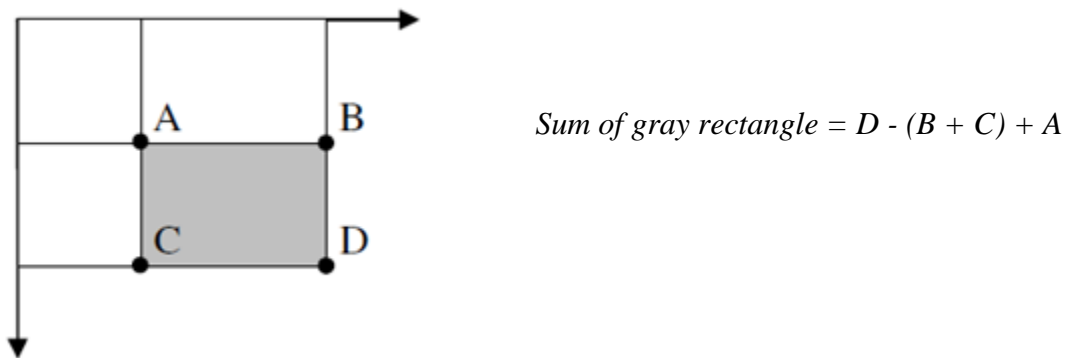


Figure 6: CALCULATION OF THE SUM OF PIXELS INSIDE ANY GIVEN RECTANGLE

Haar-like features represent characteristics of a face that often result in a large difference in intensity. For example, the vertical two rectangles features represent the eye and the cheek of a face with the eye having a higher intensity.

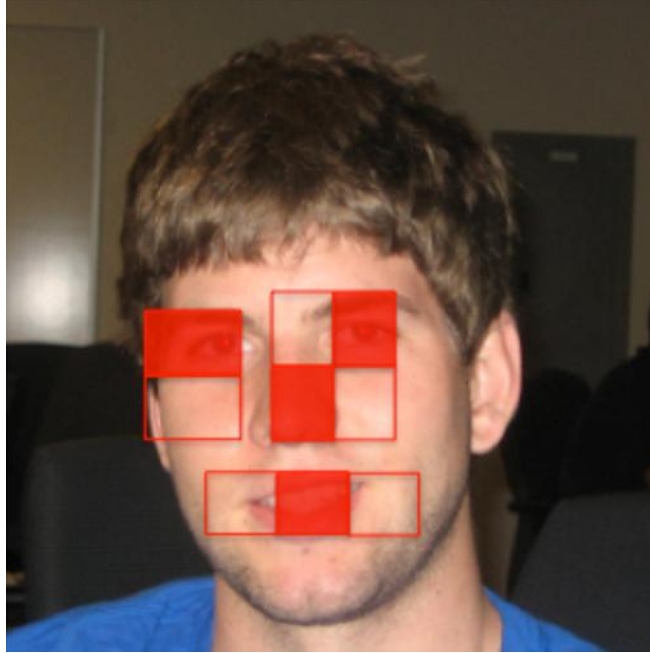


Figure 7: HOW EYES, NOSE, BRIDGE AND CHEEK ARE RELATED ON COMMON FACES

### 3.2 AdaBoost learning algorithm

Given a 24x24 search sub-window, when allowing for all possible sizes and positions of the rectangle features, a total of approximately 160.000 different features can be constructed in each sub-window. Thus, the amount of possible features vastly outnumbers the number of pixels.

Even though each feature can be computed very efficiently, computing the complete set is very expensive. It is clear that a feature selection is necessary in order to keep real-time computation time compatibility. In order to select a small number of critical features from a very large set of potential features, AdaBoost learning algorithm is used.

The AdaBoost learning algorithm combines a collection of weak classification functions to form a stronger classifier. In order for the weak learner to be boosted, it is called upon to solve a sequence of learning problems. After the first round of learning, the examples are re-weighted in order to emphasize those which were incorrectly classified by the previous weak classifier. The final strong classifier takes the form of a weighted combination of weak classifiers followed by a threshold.

A weak classifier  $h_j(x)$  consists of a feature  $f_j$ , a threshold  $\theta_j$  and a polarity  $p_j$  indicating the direction of the inequality sign:

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases}$$

The strong classifier is constructed as a linear combination of weak classifiers chosen from a given, finite or infinite, set, as shown in equation below.

$$h(x) = \begin{cases} 1 & \sum_{t=1}^T \alpha_t h_t(x) > \theta \\ 0 & \text{otherwise} \end{cases}$$

Where  $\theta$  is the stage threshold,  $\alpha_t$  is the weak classifier's weight and  $T$  the total number of weak classifiers (features).

In the Viola-Jones, a variant of Adaboost is used which performs two important tasks:

- feature selection from a huge number of potential features
- construct cascade of classifiers using selected features

### 3.3 The attentional Cascade

The third contribution of Viola-Jones algorithm is a method for combining successively more complex classifiers in a cascade structure which dramatically increases the speed of the detector by focusing attention on promising regions of the image. The notion behind focus of attention is that is always possible to determine where in an image a face might occur, more complex processing is reserved only for these promising regions.

The cascaded classifier is composed of stages each slightly more complex than the last. The task of each stage is to determine whether a given sub-window is definitely not a face or maybe a face. When a sub-window is classified to be a non-face by a given stage it is immediately discarded. Conversely a sub-window classified as a maybe-face is passed on to the next stage in the cascade. It follows that the more stages a given sub-window passes, the higher the chance the sub-window actually contains a face. The concept is illustrated with in Fig. 8.

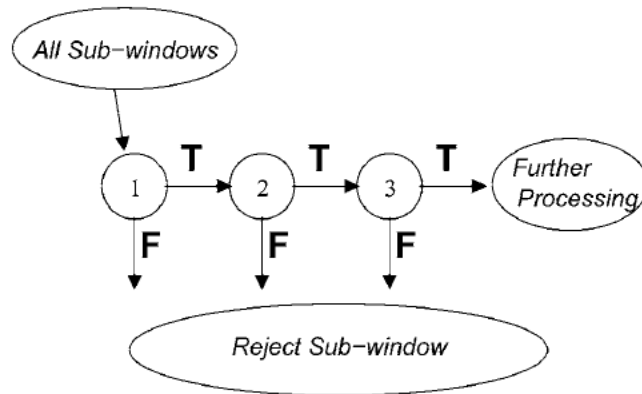


Figure 8: THE ATTENTIONAL CASCADE

The overall flowchart relative to the Viola-Jones algorithm can be seen in the Fig. 9.

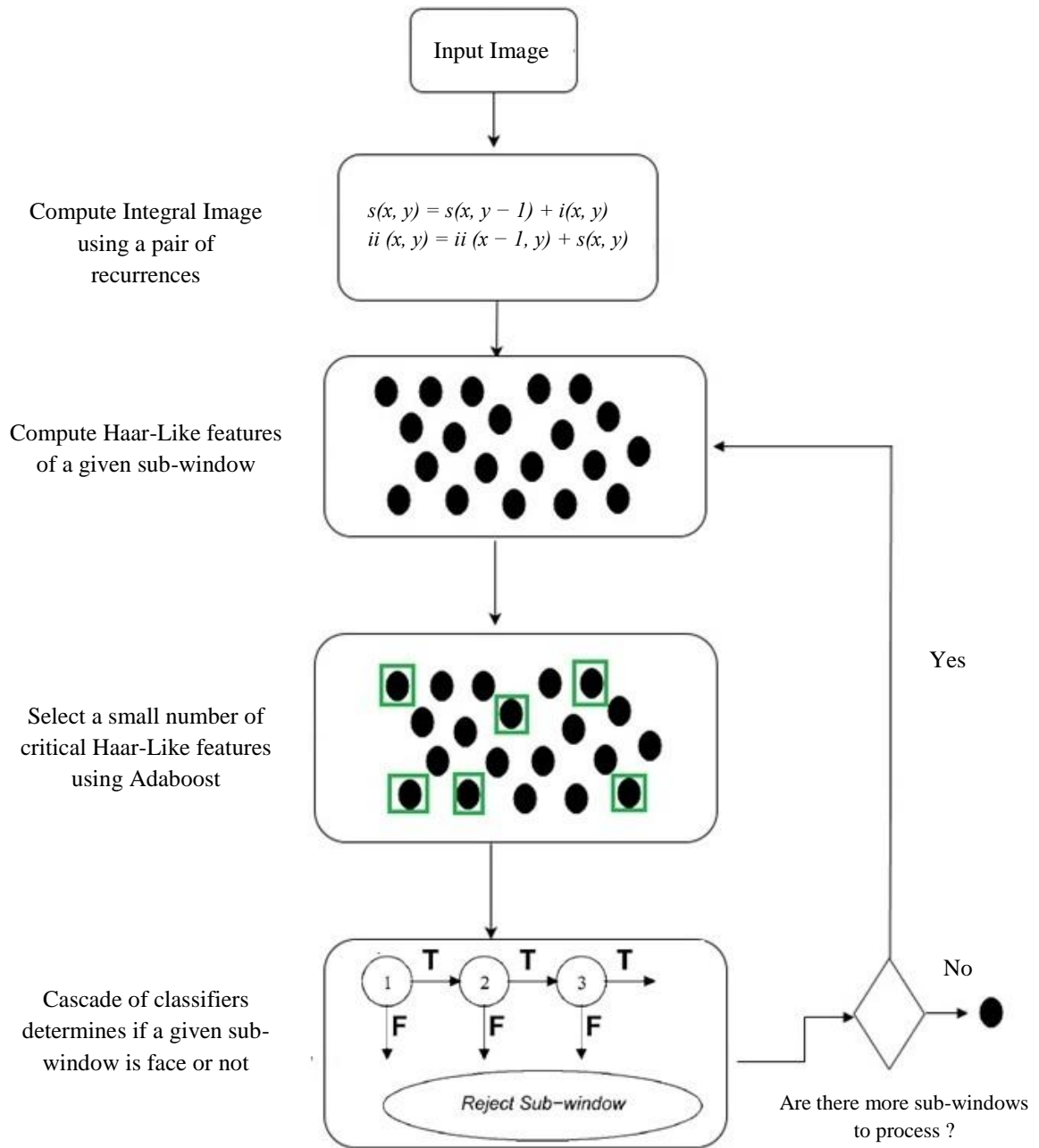


Figure 9: OVERALL FLOWCHART RELATIVE TO THE VIOLA-JONES ALGORITHM



### 3.4 Training a Cascade of Classifiers

The Viola-Jones paper also specifies how to train the cascade of classifiers. According with [6] the number of cascade stages and the size of each stage must be sufficient to achieve good detection performance while minimizing computation. Given a trained cascade of classifiers, the false positive rate of the cascade is:

$$F = \prod_{i=1}^K f_i, \quad \text{Where } F \text{ is the false positive rate of the cascade classifier, } K \text{ is the number of classifiers and } f_i \text{ is the false positive rate of the } i\text{th classifier.}$$

The detection rate is:

$$D = \prod_{i=1}^K d_i, \quad \text{Where } D \text{ is the detection rate of the cascade classifier, } K \text{ is the number of classifier and } d_i \text{ is the detection rate of the } i\text{th classifier.}$$

The overall training process involves two types of tradeoffs. In most cases classifiers with more features will achieve higher detection rates and lower false positive rates. At the same time classifiers with more features require more time to compute. In principle a possible optimization can be defined such that:

- the number of classifier stages,
- the number of features,  $n_i$  of each stage
- the threshold of each stage

are traded off in order to minimize the expected number of features  $N$  given a target for  $F$  and  $D$ .

Unfortunately finding this optimum is a very difficult problem. In practice a very simple framework is used to produce an effective classifier which is highly efficient. The user selects the maximum acceptable rate for  $f_i$  and the minimum acceptable rate for  $d_i$ . Each layer of the cascade is trained by AdaBoost with the number of features used being increased until the target detection and false positive rates are met for this level.

The speed of the cascade detector is directly related to the number of features evaluated per scanned sub-window. The number of features evaluated depends on the images being scanned.

### 3.5 Failure modes of Viola-Jones algorithm

Training a cascade with frontal, upright faces, is capable of detecting faces which are tilted up about 15 degrees in plane and about 45 degrees out of plane. The detector becomes unreliable with more rotation than this. In the cases in which the faces are very dark while the background is relatively light sometimes also causes failures. Finally, the Viola and Jones face detector fails on significantly occluded faces. If the eyes are occluded, the detector will usually fail while the mouth is not as important and so a face with a covered mouth will usually still be detected.

We propose a possible way of overcoming those failures for the case of video streams in a two step approach:

- First, apply the standard Viola-Jones face detection procedure
- Second, when the Viola-Jones procedure fails, based on the last position where a face was detected, employ a tracking algorithm which is independent of the trained cascades.

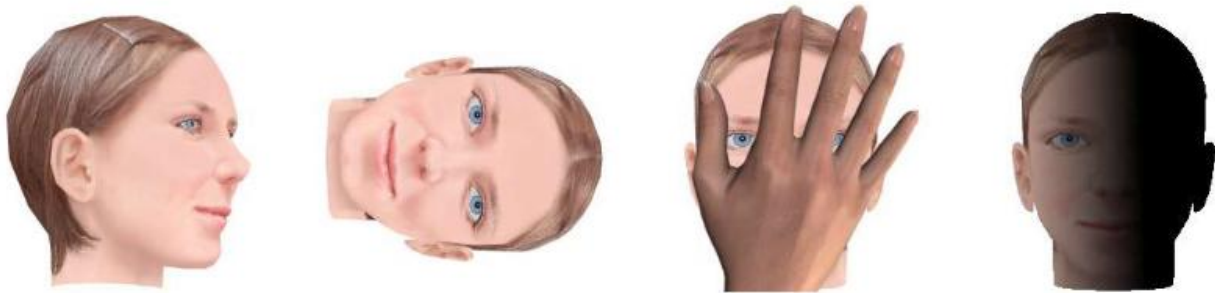


Figure 10: WHERE FACE DETECTION DOES NOT WORK

## 4. Implementation of Viola-Jones Algorithm with OpenCV

OpenCV is an Open Source Computer Vision Library originally developed by Intel. It has several implemented and optimized algorithms of image processing usage. One of those is the Viola-Jones algorithm which is capable of detecting faces and objects [7].

OpenCV applies the adaptations proposed by [8] to the original Viola-Jones algorithm. The biggest contribution of [8] was to first extend the Haar-like feature set to an efficient set of  $45^\circ$  rotated features. These novel features significantly enrich the simple features of [6] and can also be calculated efficiently. With these new rotated features the face detector shows off on average a 10% lower false positive rate.

Secondly, [8] present an analysis of different boosting algorithms (namely discrete, real and gentle Adaboost). It was proven that gentle Adaboost outperforms (with respect to detection accuracy) discrete and real Adaboost for object detection tasks, while having a lower computational complexity, i.e., requiring a lower number of features for the same performance.

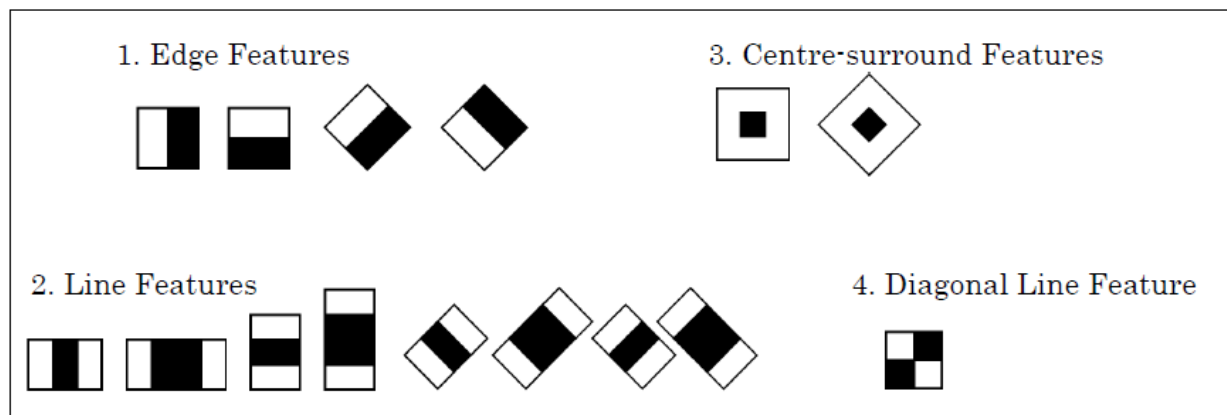


Figure 11: THE EXTENDED HAAR-LIKE FEATURE SET PROPOSED IN [8]

Even though the adaptations proposed by [8] are able to reduce the false positives rates by 10%, they are still not optimal.



Figure 12: AN IMAGE WITH FALSE POSITIVE FACES

One way of achieving very low false positives rates is to verify from the presumable detected face(s) how many facial features (eyes, nose and mouth) are present, then assign weights accordingly with the number of features present. If we are able to find all the features, we are 100% sure that a face is present.

The proposed algorithm for face detection with OpenCV is the following one:

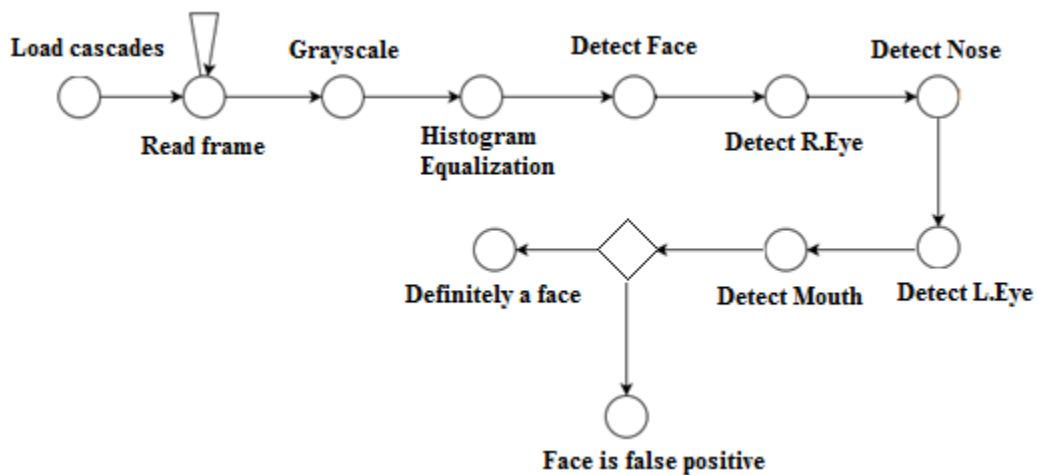


Figure 13: STAGES OF THE FACE DETECTION

```

load_cascades()

//open webcam or load a video file here

while(read_frame)
{
    convert_frame_to_grayscale()
    histogram_equalization()
    detect_faces()

    for(each_detected_face)

        right_eye = detect_right_eye()
        left_eye  = detect_left_eye()
        nose      = detect_nose()
        mouth     = detect_mouth()

        features_found = right_eye + left_eye + nose + mouth

        switch( features_found)
        {
            case 0: false positive
            case 1: 75%  of being a face
            case 2: 85%  of being a face
            case 3: 95%  of being a face
            case 4: 100% of being a face
        }
    }
}

```

## 4.1 Face Detection process

### a) Load cascades

All the already trained cascades are loaded, their format file is XML. This task is performed only once in the beginning of the execution. The cascades classifiers basically tell the application what objects to look for in images, for our specific case we are interested in the face, eyes, mouth and nose cascade classifiers.

The cascades classifiers are trained with a few hundred positive sample views of a particular object (a face, a car, a plane, etc) and negative samples, that's it, a set of arbitrary images which doesn't contain the objects we want to detect. The training process is performed offline, it doesn't belong to the object detection implementation.

### b) Convert Frame to grayscale

Converts an input frame from RGB color space to the equivalent grayscale which has a much lower complexity. Grayscale images are used on image processing applications where color information doesn't help identifying important edges or other features. The assumption is that processing a three-channel color image (RGB) takes three times as long as processing a grayscale image.

A Grayscale image allows addressing brightness, contrast, edges, shape, contours, texture, perspective, shadows, etc, without addressing color. When we are looking for patterns, as on face detection problem, and their properties such as form and shape, gray level images are sufficient.

### c) Histogram Equalization

The goal of histogram equalization is to improve the contrast in an image, in order to stretch out the intensity range [11].

Equalization implies mapping one distribution (the given histogram) to another distribution (a wider and more uniform distribution of intensity values) so the intensity values are spreaded over the whole range. Histograms are collected counts of data organized into a set of predefined bins.

The histogram equalization applies the following algorithm to the input frame:

- Calculate the histogram  $H$  for src
- Normalize the histogram so that the sum of histogram bins is 255.
- Compute the integral of the histogram
- Transform the image using  $H'$  as a look-up table:  $dst(x, y) = H'(src(x, y))$

The algorithm normalizes the brightness and increases the contrast of the image. It is very difficult to achieve perfect histogram equalization, but there are various techniques to achieve histogram equalization close to the perfect one such as the one defined in OpenCV [12].

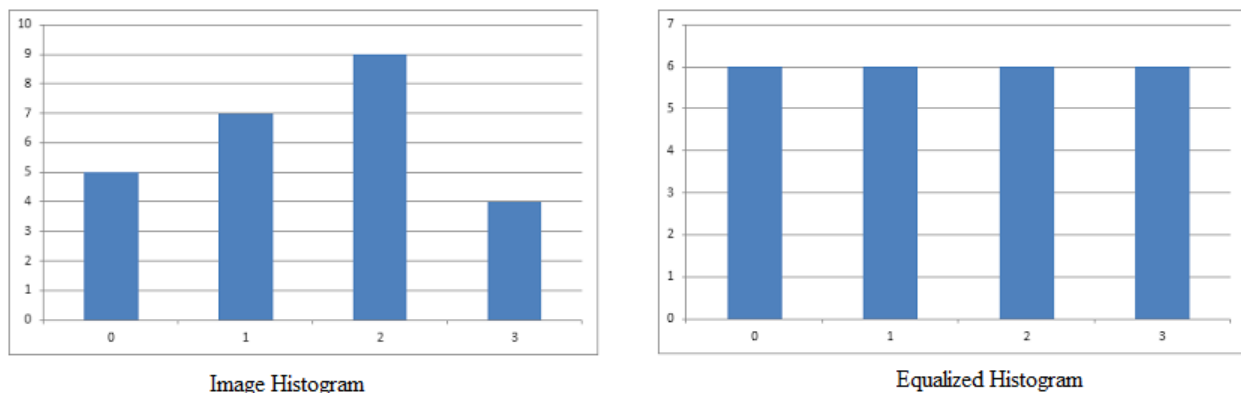


Figure 14: PERFECT HISTOGRAM EQUALIZATION

The histogram equalization described above considers the global contrast of the image. In some cases, it is not a good idea because we can lose a lot of the information due to over-brightness. This happens because the histogram is not confined to a particular region [13].

**CLAHE (Contrast Limited Adaptive Histogram Equalization)** differs from ordinary histogram equalization in the respect that the adaptive method computes several histograms, each corresponding to a distinct section of the image called tiles (tile-size is 8x8 by default). It is therefore suitable for improving the local contrast and enhancing the definitions of edges in each region of an image.

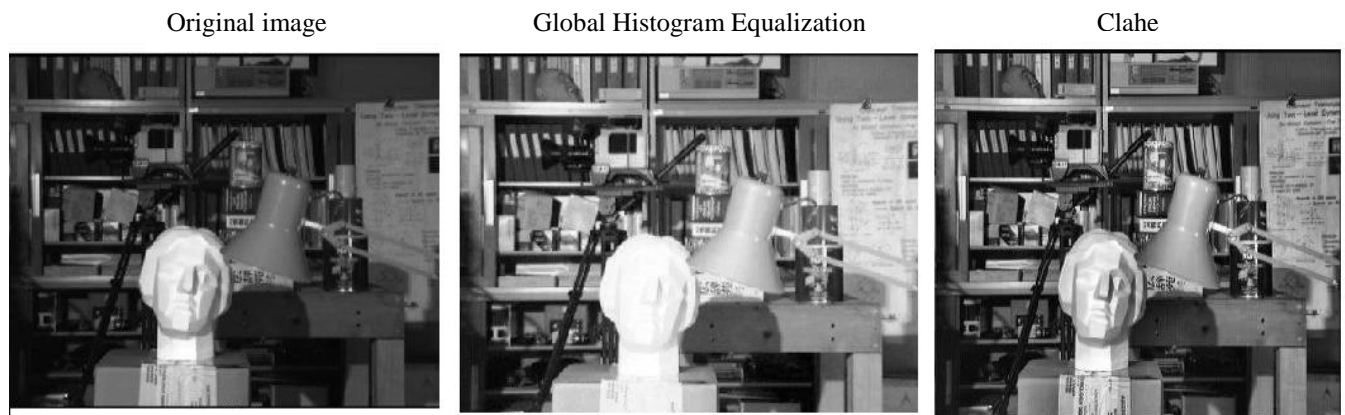


Figure 15: HISTOGRAM EQUALIZATION VS CLAHE

Now a question that comes to mind is which histogram equalization method should we apply? The answer to that question is not simple since both of them are optimal in different scenarios. It is important to point out is that the clahe method computation time is on average 3 times more expensive than the global histogram method (`equalizeHist`).

The proposed solution to this problem is the following one:

1. Start by defining a threshold for both methods. Taking into account that the computation time of clahe is 3 times more expensive than the `equalizeHist`, the thresholds could be defined in such a way that  $equalizeHist\_threshold = x$  and  $clahe\_threshold = y$  in which  $x < y$
2. By default the first frame applies the `equalizeHist`
3. If faces are detected, then set the counter to zero
  - 3.1 next frames must apply the same equalization method
4. If faces are not detected increment the counter
  - 4.1 if counter is less than the threshold go to point 3.1
  - 4.2 if counter is greater than the threshold switch equalization method, set counter to zero and start again from point 3



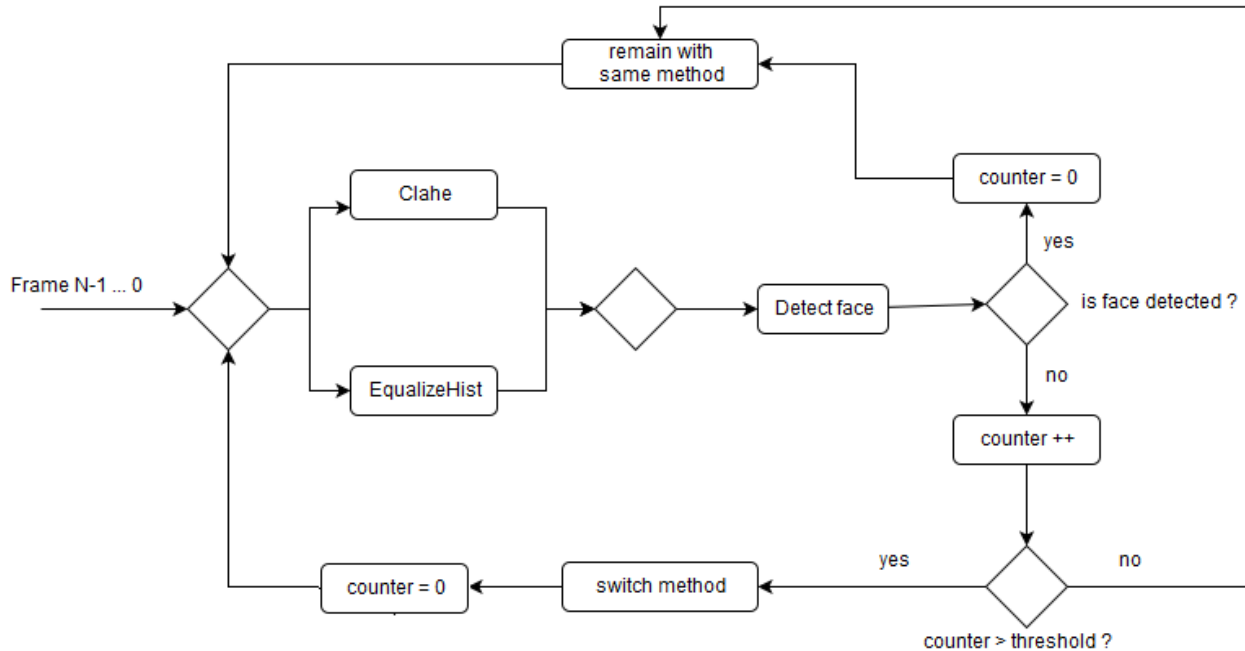


Figure: 16 HISTOGRAM DIAGRAM

#### d) Face Detection

The main goal of this task is to find face(s) in a static image or video frame. In order to do so, each sub-window of the input frame or image is matched against the information contained in the cascade of classifiers. When the task finishes, it returns the face(s) extent across the input, if any. A detailed description of this task is given on section 5.1.

#### e) Facial Features Detection

When detecting features of a possible face, one important aspect to take into account is the concept of Region Of Interest (ROI) which allows to split the face in parts in order to reduce the computation time. ROI enables to compute each feature on a very small sub-window. For example if we want to verify if a mouth is present on a face, we do not need the entire face but only the bottom-center part of it.

It is also important to notice that when detecting the eyes, we need to determine if they are open or not. There are several methods to achieve this goal such as:

- Apply skin detection in the region of the eye, if the percent of skin pixels is greater than a threshold, then the eye is closed
- Look for circles in the eye ROI, if found, the eye is open, otherwise the eye is closed
- Apply two cascades, first the one that recognizes both open and closed eyes, secondly apply a cascade that recognizes only open eyes. If the second cascade returns a positive result the eye is open, otherwise the eye is closed



By experiment, was possible to notice that the last method is the most efficient to determine if eyes are either open or closed.

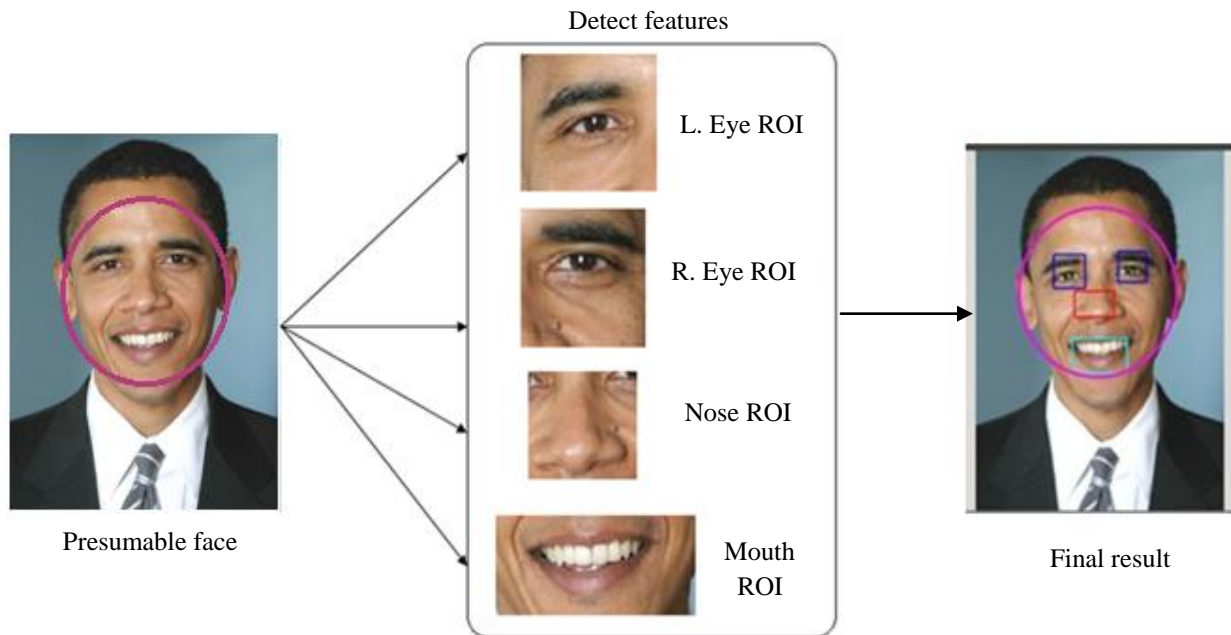


Figure 17: CONCEPT OF ROI

## 4.2 Face Tracking

As described in Section 3.5, the Viola-Jones algorithm still has some failure modes such as:

- the detector becomes unreliable for faces with rotation greater than 15 degrees in plane and 45 degrees out of plane
- the detector sometimes fails when the faces are very dark while the background is relatively light
- the detector fails on significantly occluded faces, if the eyes are occluded, the detector will usually fail while the mouth is not as important

We would like to implement a solution capable of overcoming those failure modes. Our solution is based on a two step approach which are:

- First, apply the standard Viola-Jones face detection procedure
- Second, when the Viola-Jones procedure fails, based on the last position where the face was detected, employ a tracking algorithm which is independent of the trained cascades.

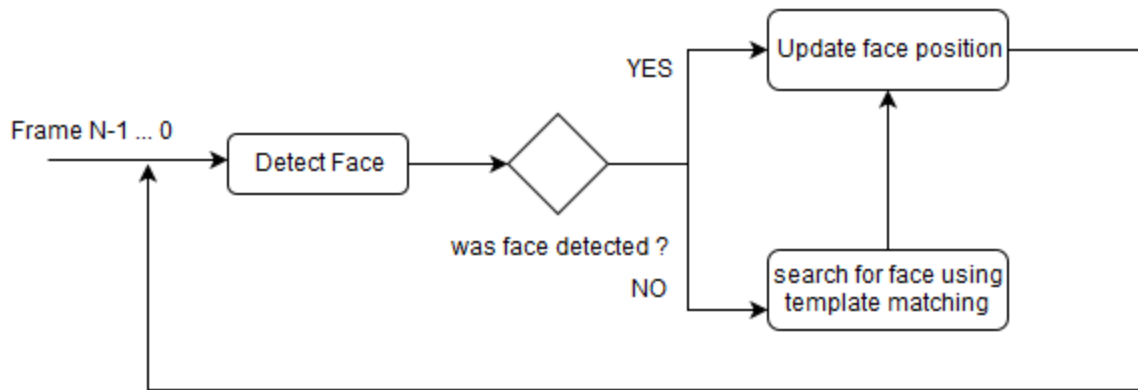


Figure 18: FACE TRACKING DIAGRAM

One could argue why not after detecting a face on frame  $i$ , employ a tracking algorithm and remain with it without re-applying the detecting procedure on frame  $i+1$ . The reason is simple, even the best tracking algorithm available cannot detect a face precisely as the face detection procedure. So the best way is to work with a tracking algorithm as a backup.

#### 4.2.1 Evaluation of Existing Object Tracking Algorithms

Tracking is the ability to follow an object through a sequence. On a tracking system, it is essential to exploit the temporal correspondence between frames in order to incorporate the object changes over time, in terms of changes in scale, position and to localize the search for the object [14].

##### Camshift (Continuously Adaptive Mean-Shift)

There are a lot of object tracking algorithms available. Camshift is an algorithm which is very good for single hue object tracking in the condition where object's color is different with respect to the background's color [14].

The Camshift algorithm can be summarized as follows:

1. First we need to choose the initial region of interest containing the object that we want to track.
2. Make a color histogram of that region as the object model.
3. Apply back projection, a method which using the histogram of an image shows up the probabilities of the colors that may appear in each pixel.
4. Apply MeanShift, an algorithm which finds modes (the values that appears most often) in a set of data samples representing an underlying probability density function. The algorithm work as follows:

- a) Initialize the sphere, including the center and radius.
  - b) Calculate the current mass center.
  - c) Move the sphere's center to mass center
  - d) Repeat step b and c, until converge, that is, current mass center after calculate, is the same point with center of sphere.
5. Track using the MeanShift algorithm to every single frame, and the initial window of each frame is the output window of the prior frame.

Although being very effective, Camshift algorithm does not consider an object's direction vectors and velocity. In addition, sometimes, Camshift loses the tracked object when the color of the background is similar to that of the object.

### Camshift with Kalman filter

In order to solve the above ambiguities, [15] proposes to combine Camshift with Kalman filter, an algorithm that estimates the position of the object in each frame of the sequence. The Kalman filtering used in object tracking is an optimal estimation method with the criterion of minimal error covariance.

The combination of Camshift with Kalman filter algorithm start by:

1. Getting the input image
2. Input image is converted to remove noise.
3. Camshift with the Kalman filter is used to track the object

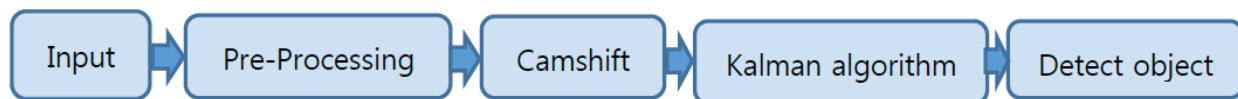


Figure 19: CAMSHIFT WITH KALMAN FILTER ALGORITHM

The advantage of this combination is that provides a low calculation scale and high real-time performance. However, it supposes a linear Gauss state space model, which may be not consistent with the motion situation of the object in the real world.

### Template Matching

On this object tracking algorithm, the generated templates from each frame are passed on to the tracking module, which starts tracking the object with an input reference template. The module uses template-matching to search for the input template in the scene grabbed by the camera [24].

A new template is generated if the object is lost while tracking due to change in its appearance. It is also important to point out that generations of such templates are dynamic which helps to track the object in a robust manner.

There are several types of template matching methods such as the sum of squared difference, cross correlation and coefficient correlation. According with the studies made by [25] which compared the methods used in template matching, the best one is the correlation coefficient method.

Template matching becomes inefficient, as calculating the pattern correlation image for medium to large images is time consuming.

#### 4.2.2 Considerations about the object tracking algorithms

As it is possible to verify from the Fig.20, both Camshift with Kalman filter and Template Matching algorithms are able to track a face with rotation greater than 15 degrees in plane and 45 degrees out of plane which is one the drawbacks of the Viola-Jones algorithm.



Figure 20: LEFT IMAGE CAMSHIFT WITH KALMAN FILTER, RIGHT IMAGE TEMPLATE MATCHING

For the overall different scenarios that were tested, Template Matching clearly demonstrated better results. In order to overcome its inefficiency, as it becomes time consuming for medium to large images, only the area around the nose is used as input template to be matched with next frames.

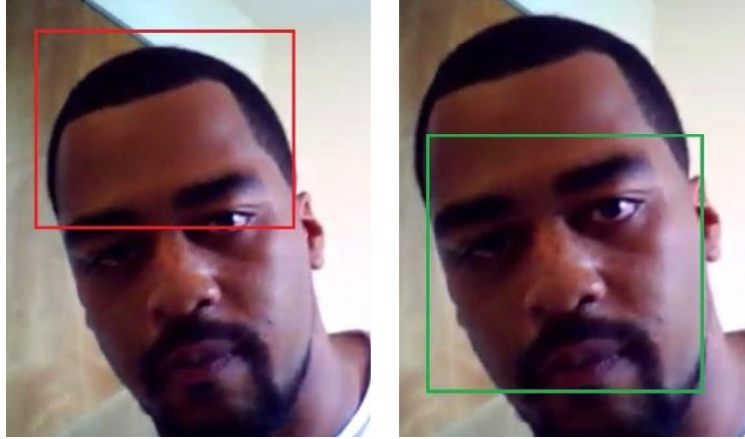


Figure 21: LEFT IMAGE CAMSHIFT WITH KALMAN FILTER, RIGHT IMAGE TEMPLATE MATCHING

While we are able to overcome most of the failure modes described in Section 3.5, the tracking algorithm is still not optimal as it first needs the Viola-Jones algorithm to know what to track, so before knowing that, we are still limited to the bounds of the Viola-Jones algorithm.

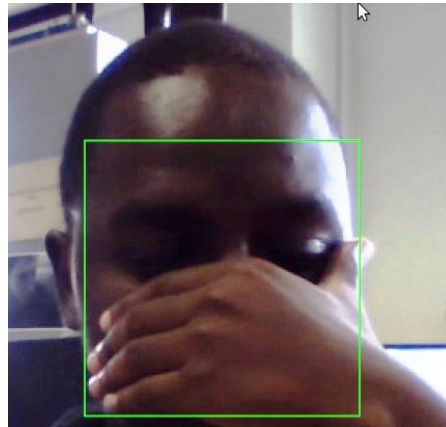


Figure 22: THE TRACKING WORKS EVEN WHEN A PART OF THE FACE IS OCCLUDED

## 5. Sequential Implementation

The sequential implementation of the face detection algorithm is important to know the exactly the time taken by each task. From the results obtained, as is possible to see from the figures below (Fig 23, 24 and 25), it is clearly that the finding face is the most expensive task.

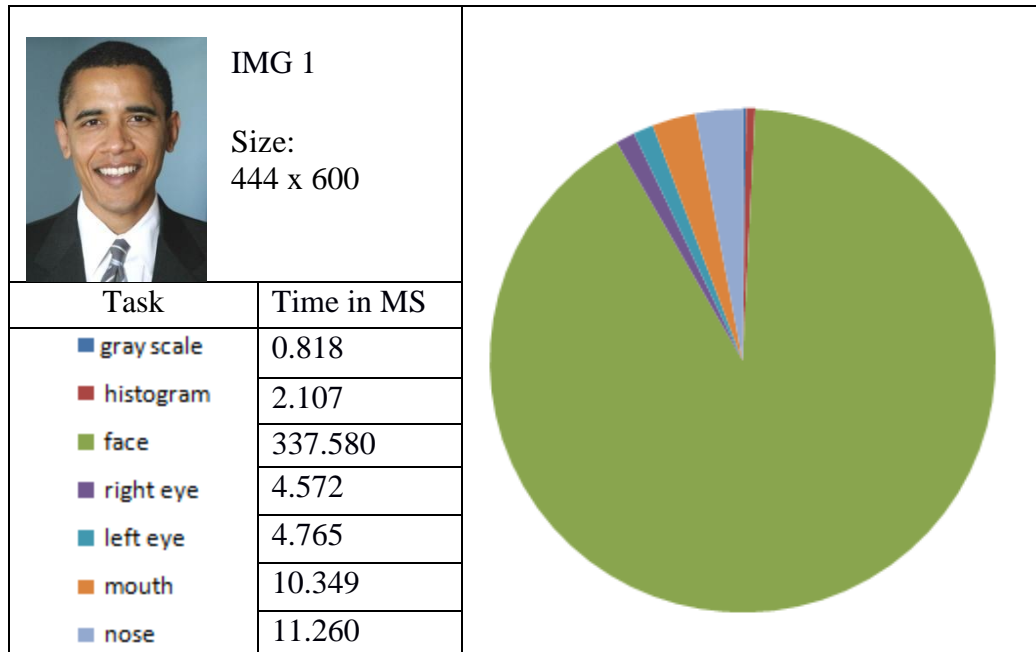


Figure 23: IMG 1

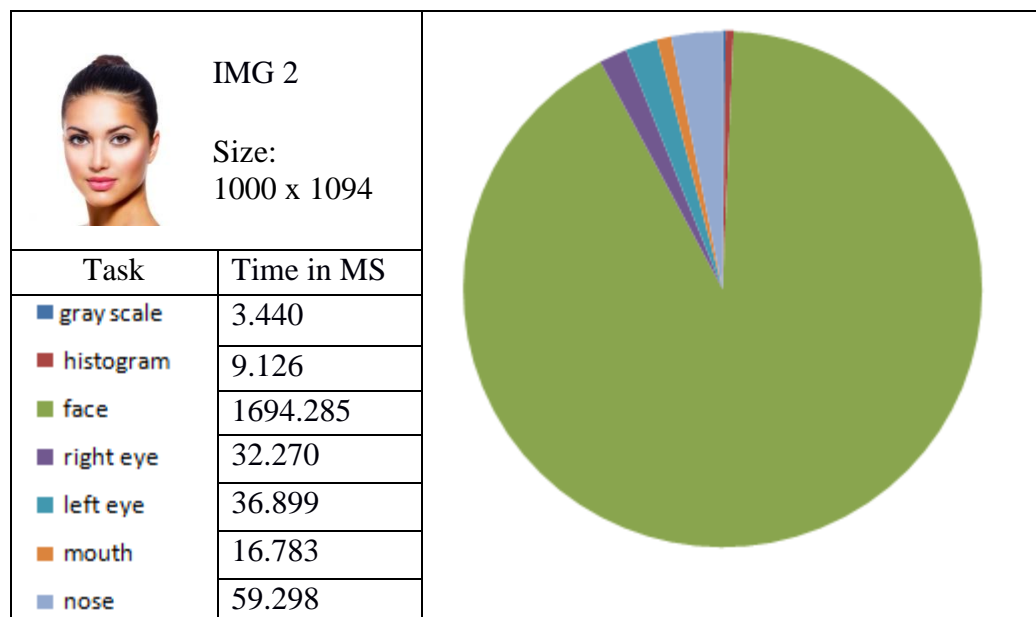


Figure 24: IMG 2

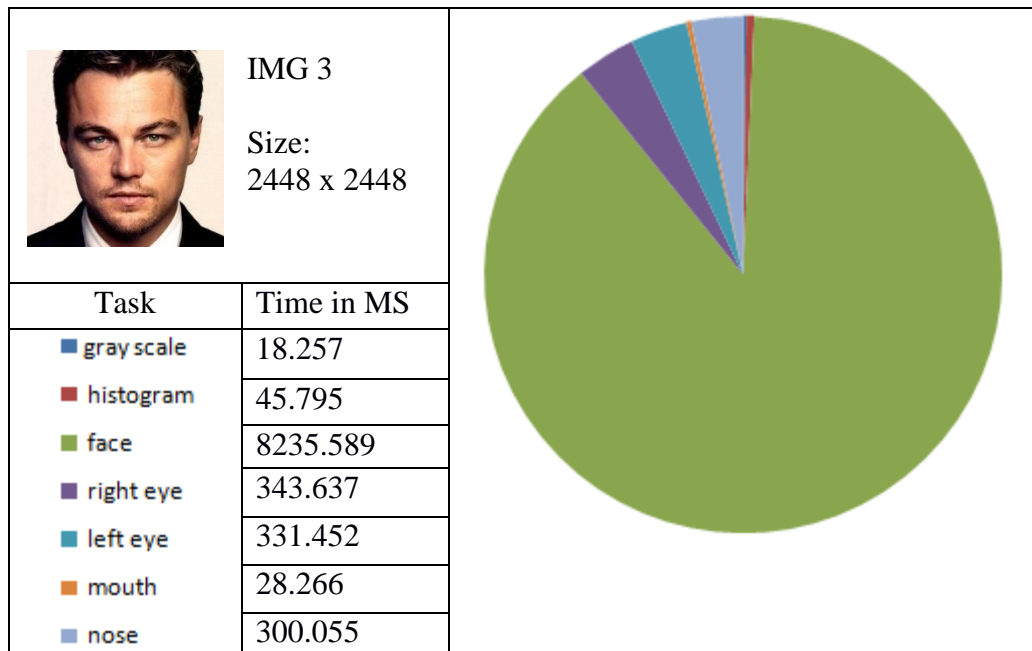


Figure 25: IMG 3

## 5.1 Optimizations

### 5.1.1 Face detection optimization

We would like to optimize the face detection task in order to reduce its computation time. To do so it's important to understand how it works inside.

The face detection task first creates an image pyramid which is a multi-scale representation of the image such that the face detection can be scale-invariant, that's it, detecting large and small faces using the same detection window (sub-window).

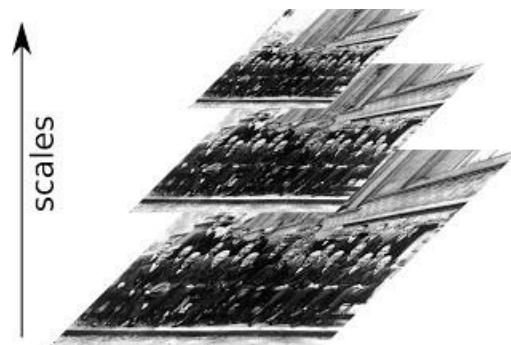


Figure 26: PYRAMID OF SCALES

On each scale, it is applied the overall flowchart described in section 3.3. It's important to notice that the size of the detection window is the same in all the scales. This process is such that smaller faces are detected on early scales while bigger faces are detected later.



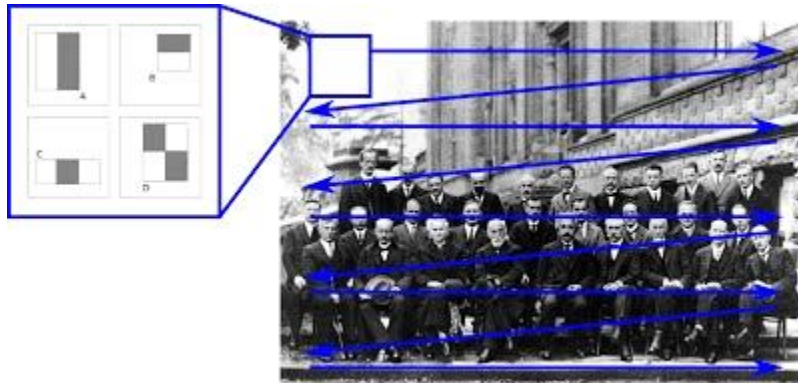


Figure 27: DETECTION WINDOW (SUB-WINDOW)

To optimize this process it's important to answer some questions:

- How many scales are needed?
- What should be the size of the detection window?

To answer those questions it is important to define some parameters first.

**Scale factor:** specifies how much the image size is reduced at each image scale. It allows the creation of the image pyramid. By rescaling the input image, we can resize a larger face to a smaller one, making it detectable by the algorithm.

1.10 is a good possible value, which means each scale will reduce the size by 10% with respect to the previous one. This also means that the algorithm works slower since a lot of scales are created. It is possible to increase it to as much as 1.40 for faster detection, with the risk of missing some faces altogether.

In order to precisely detect the face, all the scales that return a face result are merged into one final result.

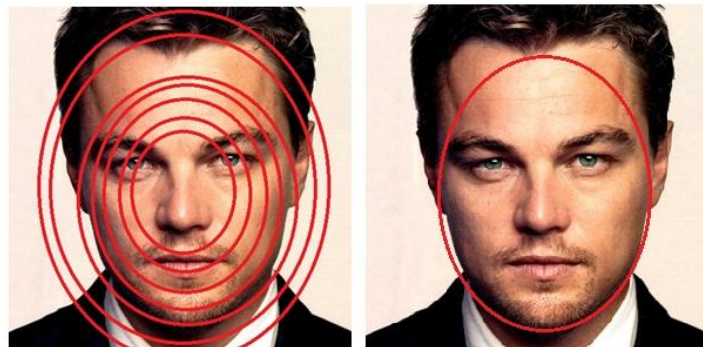


Figure 28:

46 scales were generated. scales [1-35] and [43 - 46] did not find any face, scales [36 – 42] did find a face (left image). all the scales that found a face are merged into one final result (right image).



**Minimum Object Size:** minimum possible object size, objects smaller than that are ignored. It determines how small is the object size that we want to detect. The standard for this parameter is usually [30, 30] pixels.

**Maximum Object Size:** maximum possible object size, objects bigger than this are ignored.

Now that the parameters are known, we would like to work with low scale factor (1.1) so that all the possible faces are detected but also with a few numbers of scales. The stopping condition for the creation of scales is based on the two parameters defined above (min and max object size).

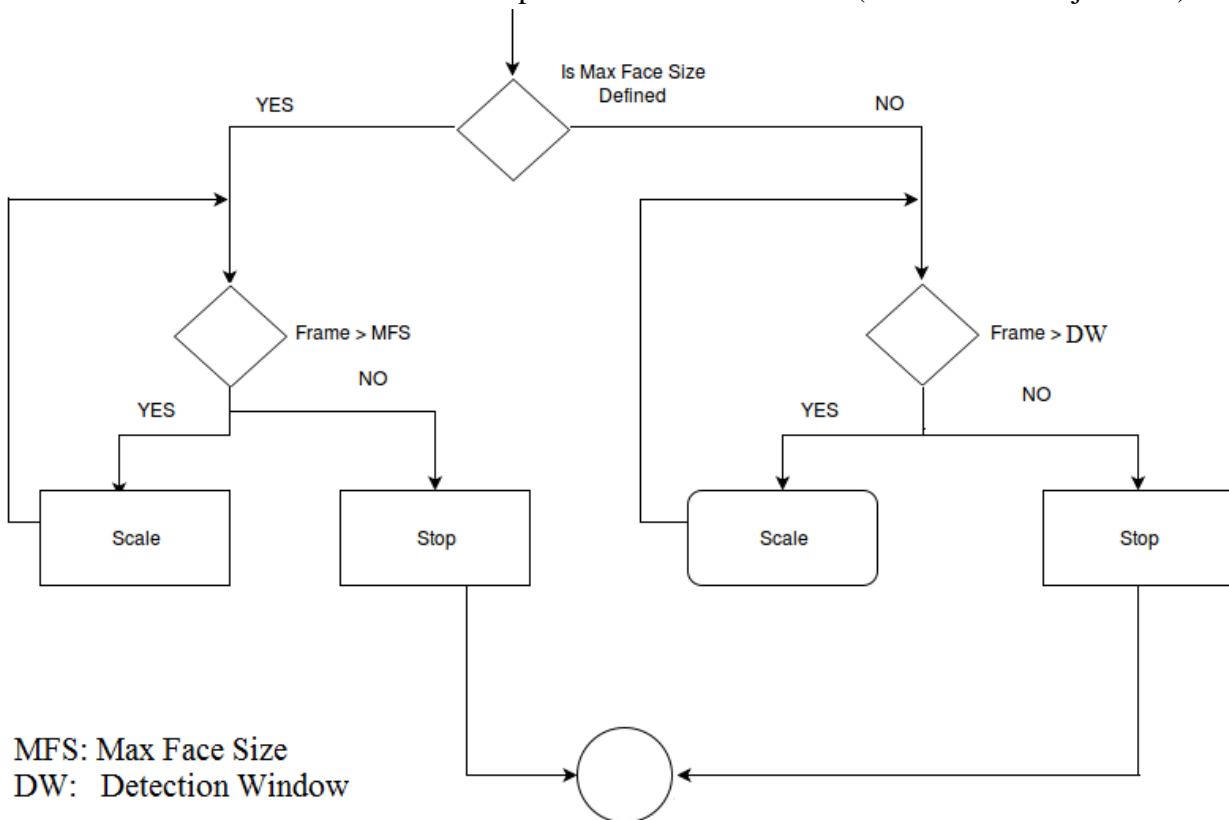


Figure 29: STOPPING CONDITION FOR THE CREATION OF SCALES

The size of the detection window is defined as the double of the min object size. So only objects that are greater than the min object size and smaller than the detection window are considered.

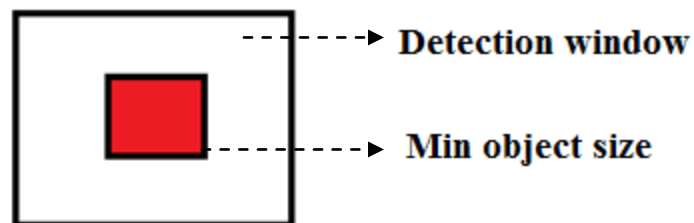


Figure 30: DETECTION WINDOW

By experiment was possible to notice that the most important factor to optimize the face detection is the min object size, as it directly affects the size of the detection window and the number of scales.

A final optimization solution for a stream of images can be seen as:

1. first we start by considering the worst case scenario in which the parameter for min face size is (30, 30) and a scale factor of 10% which will generate a lot of scales
2. once a face is found, adjust the detection window to the size of the face for the next frames ( $min\_face\_size = real\_face\_size \times 0.8$ )

This process allows reducing drastically the number of scales as they are determined based on the previously detected face dimensions as it's possible to see from the Fig. 31.

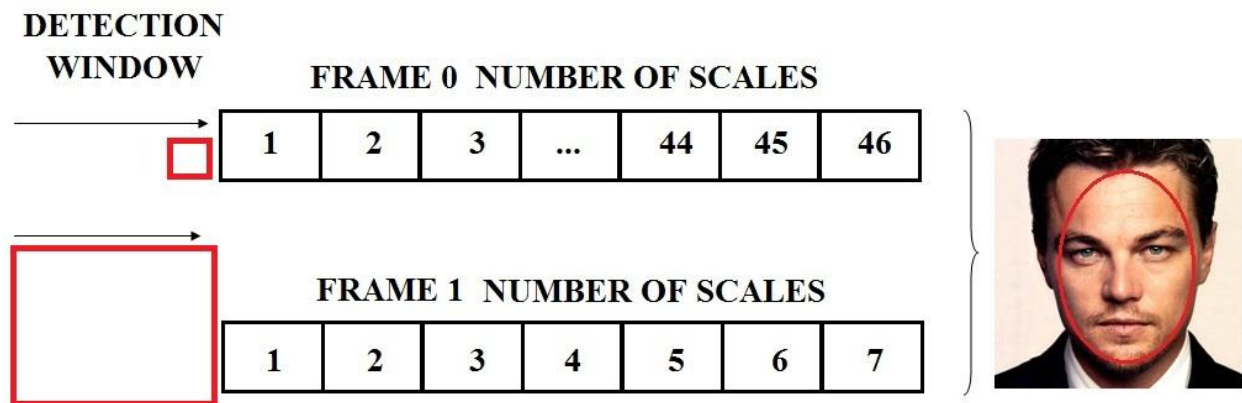


Figure 31: OPTIMIZED SOLUTION

As it is possible to notice from the Fig 32, the computation time decrease drastically as the min face size gets closer to the real face size and the number of scales are reduced.

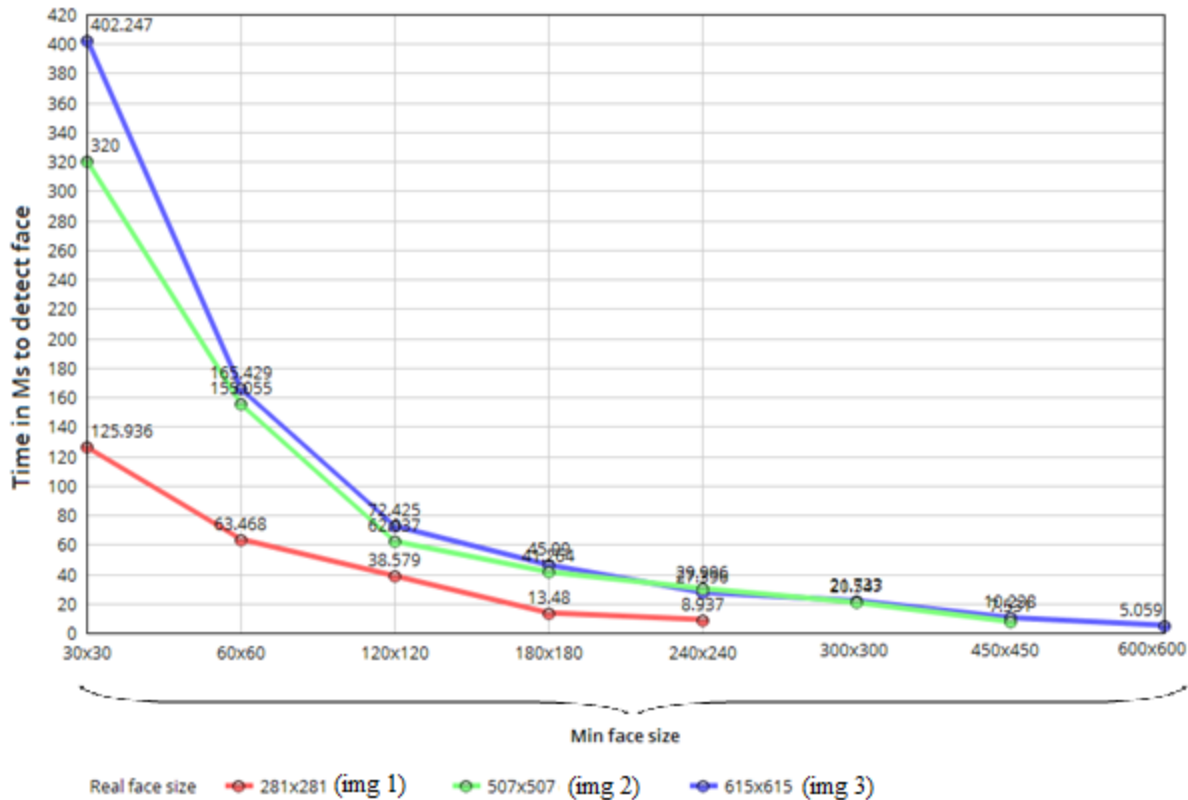


Figure 32: COMPUTATION DECREASED AS THE MIN FACE SIZE GETS CLOSER TO THE REAL FACE SIZE

This face detection optimization should be applied only for the cases where the goal is to find exactly one face or at most multiple faces that are somehow equidistant from the camera. As it's possible to notice from the Fig. 33, when the algorithm is applied for the first time (left image), we do not have any a priori knowledge of the size of the faces and almost all them are found.

Because of the image pyramid of scales, the smallest faces (the ones much more far away from the camera) are the first to be found. Then the intermediate faces (the ones on the center) and finally the largest faces (the ones in the front line).

In this way, the largest faces are the last ones to be found, so they will set the threshold of what size of face to look for. When the algorithm is applied for the second time (right image), basically, only the front line faces are found.



Figure 33: WHERE THE OPTIMIZATION SHOULD NOT BE APPLIED

### 5.1.2 Feature detection optimization

Another optimization born by experiment is that when the presumable face is found, it is possible to predict the minimum size of the features (nose, eyes and mouth) to look for. This optimization also allows reducing drastically the features computation time.

Based on worst case analysis the minimum size of the features with respect to the face are:

- **eyes:** 15% x 15%
- **nose:** 18% x 18%
- **mouth:** 30% x 15%

Features	IMG 3		IMG 2		IMG 1	
	Face size: 615x615		Face size: 507x507		Face size: 281x281	
	Real size	Worst case	Real size	Worst case	Real size	Worst case
Right Eye	288x288	92x92	124x124	76x76	57x57	42x42
Left Eye	250x250	92x92	135x135	76x76	56x56	42x42
Nose	292x350	111x111	95x115	91x91	58x70	51x51
Mouth	544x326	185x92	198x119	152x76	97x58	84x42

Table 1: Feature Optimization

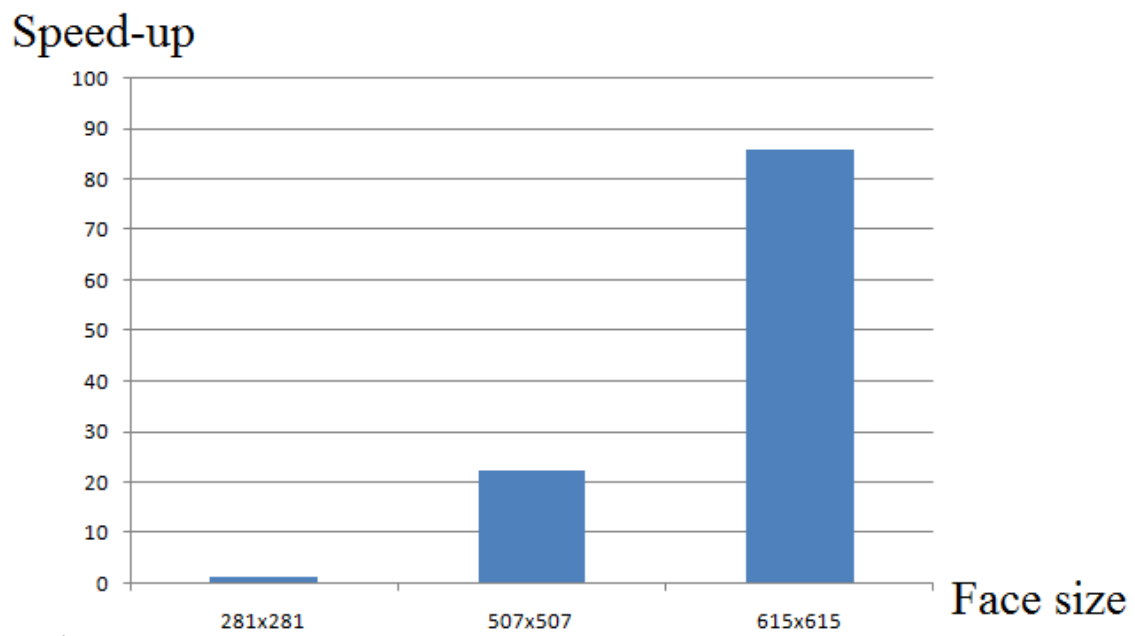


Figure 34: SPEED UP ACHIEVED WITH FEATURES OPTIMIZATION

## 6. Parallel Implementation

In this section, we will start by studying the parallel patterns and tools that will be applied later on in order to parallelize the application. The potential sources of parallelism of the Viola-Jones algorithm will be also analyzed followed by a detailed study of the OpenCV parallel implementation. An improvement to the OpenCV parallelism will also be addressed along with the GPU implementation of Viola-Jones algorithm.

### 6.1 Parallel Patterns and Tools

In this section we will discuss techniques for the design of parallel computations and the tools that allow implementing such techniques. The main sources of reference for this section are [16] and [17].

#### Pipeline Paradigm [16]

This paradigm is defined for stream of computations only. It is applied when we have a computation that can be expressed as a composition of functions. The service time (the time between the delivery of two consecutives results) of the pipeline depends on the slowest stage.

This paradigm is generally characterized by a latency (time between input of a data and the output of its result after processing) worst than the sequential version due to the communication between the stages.

The Pipeline may be used to improve the service time, however, in order to remove application bottlenecks with this paradigm we must ensure that the number of functions is equal or greater than the optimal parallelism degree and the stages are somehow balanced.



Figure 35: PIPELINE

### Embarrassingly Parallel Paradigm or Farm [16, 17]

This is a stream paradigm which replicates the same function such that distinct stream elements can be processed by distinct identical modules (workers) in parallel. This paradigm is composed by three modules: the Emitter, the Workers and the Collector.

The emitter provides to send every received input stream to one of the workers according to a certain policy. Each input element is transmitted to exactly one worker. Each worker applies the function  $F$  to the received data and sends the result to the collector which transmits each of them to the output stream.

The Farm service time is given by the maximum among the time spent by the emitter to schedule a task to one of the workers, the time spent by one of workers to process a task divided by the number of the workers in the farm and the time spent by the collector to gather a result from one of the workers and to deliver it to the output stream.

In order to apply the farm paradigm the function  $F$  must be a pure function, that's it without an internal state. The farm paradigm is good for applications where the order of the output is not so important as the relative speed of the workers can be different.

In Fig. 36, we can see the farm as a three stage Pipeline, in which the emitter is the first stage, the workers the second and the collector the last stage.

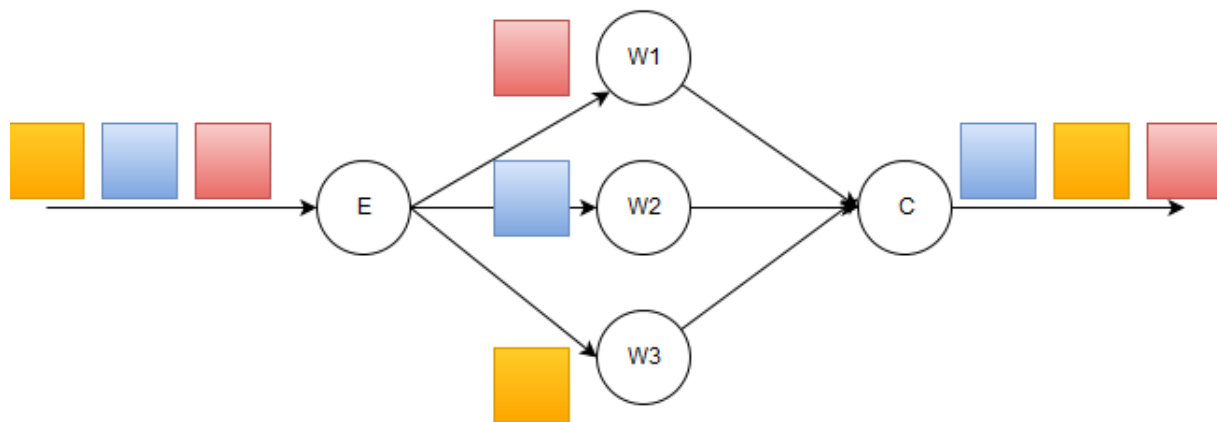


Figure 36: EMBARRASSINGLY PARALLEL OR FARM

### Map or Parallel For paradigm [16, 17]

This is a data parallel paradigm which can be applied on streams as well on a single data values. This paradigm is characterized by data partitioning and by function replication. For stream-based computations, besides service time, it also optimizes latency and memory capacity which is not the case with the paradigms introduced above.

In the Map paradigm workers are fully independent which means that each of them operates on its own local data only, without any cooperation with the other workers during the execution of the function F.

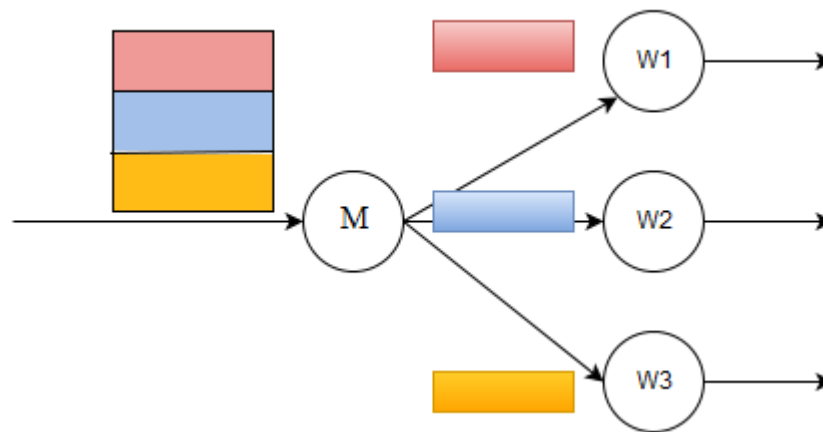


Figure 37: MAP PARADIGM

Parallel for is a paradigm characterized by parallelizing sequential iterative for loops with independent iterations. Parallel for loops may be clearly parallelized by using the map or farm patterns, but this typically requires a substantial re-factoring of the original loop code [29].

### Fast Flow (FF)

FastFlow is a C++ parallel programming framework which focuses on high-level, pattern-based parallel programming [31]. It supports streaming and data parallelism. This parallel programming framework provides application designers with key features for parallel programming with the help of suitable parallel programming abstractions and a carefully designed run-time support.

The FastFlow provides several pre-defined, general purpose, customizable and composable parallel patterns such as Pipeline, Farm, and Map. Composable means that a pattern may be used as actual parameter for other pattern. It allows expressing the combination of different patterns which increases the parallelism exploitation possibilities [17].



## OpenMP

OpenMP is a programming framework that greatly simplifies writing multi-threaded programs in Fortran, C and C++ [30]. This programming framework provides loop parallelization with parallel for.

Besides being able to implement parallel loops when iterations are independent, it also has the notion of task parallelism which allows assigning different dependent or independent tasks to different threads.

OpenMP consists of a set of compiler directives (pragmas) that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism [32].

In order to write efficient programs with OpenMP, we need to understand all the additional parameters of the OpenMP pragmas and also the concurrent accesses performed on shared data structures.

## 6.2 Parallel implementation of OpenCV

In order to implement parallelization, OpenCV uses a hierarchy of tools. The first tool found in the user machine is the one that is going to be applied.

1. HAVE\_TBB - 3rdparty library, should be explicitly enabled
2. HAVE\_CSTRIPES - 3rdparty library, should be explicitly enabled
3. HAVE\_OPENMP - integrated to compiler, should be explicitly enabled
4. HAVE\_GCD - system wide, used automatically (APPLE only)
5. WINRT - system wide, used automatically (Windows RT only)
6. HAVE\_CONCURRENCY - part of runtime, used automatically (Windows only - MSVS 10, MSVS 11)
7. HAVE\_PTHREADS\_PF - pthreads if available

For our specific case, all the parallelization implemented by OpenCV was performed with OpenMP.

For the object detection task, OpenCV automatically parallelizes the data inside each scale with a row wise Parallel for in which divides the computation space into a collection of detection windows (see Fig.38).

Each detection window is calculated independently without any need for synchronization. If the user doesn't specify the number of threads to be used, OpenCV will automatically use all of the available threads.

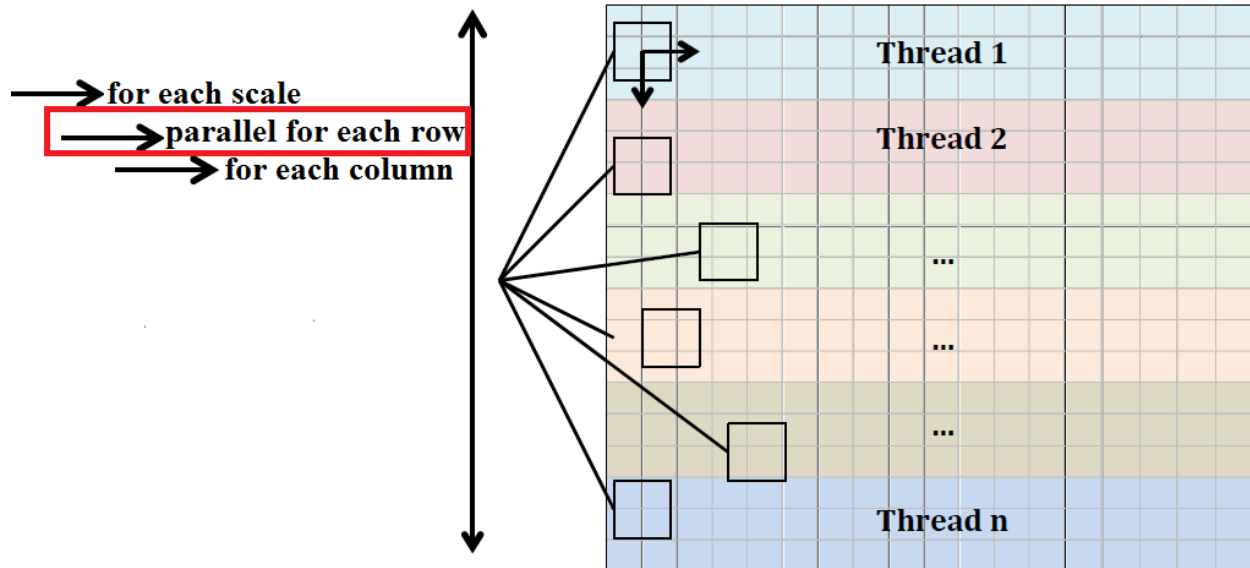


Figure 38: OPENCV PARALLELIZATION INSIDE EACH SCALE

Although is not specified in the OpenCV documentation, as it will be possible to see in Section 7.2, the tasks convert image to gray scale and perform histogram equalization are also parallelized internally.

### 6.3 Sources of Parallelism

In this section, we will start proposing our parallel implementation. In order to determine the set of activities that can be executed in parallel, we may start by looking for the all concurrent activities we can figure out in our application.

When considering the Viola-Jones algorithm described in previous sections, we have plenty of possible sources of parallelization. We will first discuss the pre-processing phases.

**Convert an RGB image to a grayscale:** there is no data dependency between the three channels composing an RGB image when performing the grayscale conversion. In this way, we can exploit the Map paradigm to parallelize this task. Synchronization is needed in order to have the equivalent grayscale of each channel. The final step is to merge the three channels grayscale images into a single one.

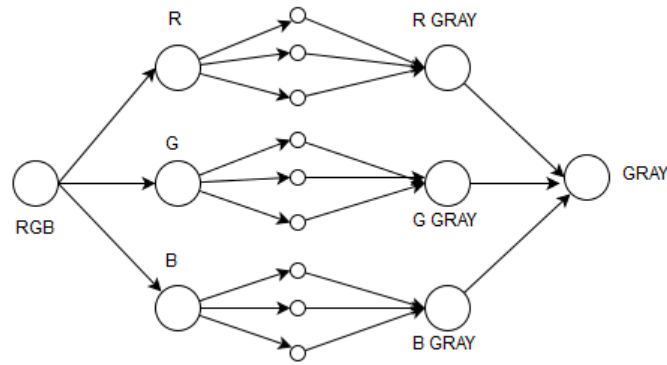


Figure 39: CONVERT AN RGB IMAGE TO A GRAYSCALE

**Histogram Equalization:** when performing histogram equalization (Section 4.1), it's possible to exploit the Map paradigm in which the image is split in order to compute the histogram  $H$  of  $src$ . A synchronization point is needed in order to have the final histogram  $H$ .

The computation of the histogram integral image is performed using a pair of recurrences as described in Section 3.1. Transforming  $H'$  as a look-up table can also be performed independently by applying the Map paradigm.

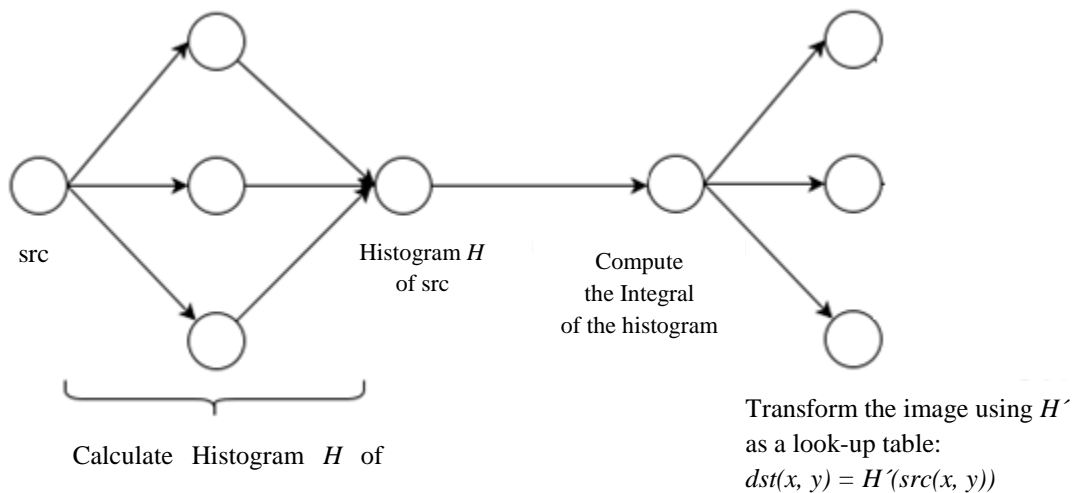


Figure 40: HISTOGRAM EQUALIZATION

Now we will move to the Viola-Jones algorithm parallelism discussion.

**Detection window:** it is possible to divide the computation space of face detection into a collection of detection windows in which each can be calculated independently without any need for synchronization. The schema for this case is the same of the Fig. 37 showed for the Map paradigm in which the input image is partitioned and detection windows would be applied in parallel in each of the partitions.

**Haar-Feature Calculation and selection:** inside each detection window, there is absolutely no data dependence among the feature value calculations and selection in such a way that each feature can be calculated and selected in a completely independent way. All the selected features are sent to the cascade of classifiers (see Fig 41).

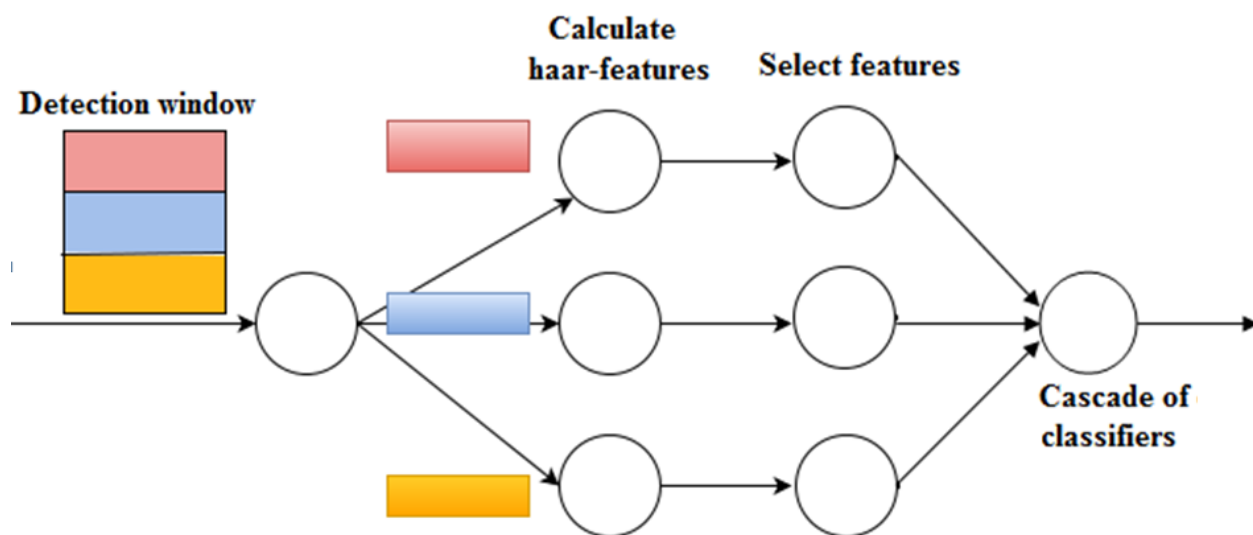


Figure 41: HAAR-FEATURE CALCULATION AND SELECTION

**Feature Evaluation:** all the selected features are evaluated by the stages of the cascade classifiers which can be parallelized using the pipeline paradigm.

Inside the stages of the cascade classifiers, the evaluation process is independent and each feature must go through all the filters of the stage. This process is suitable for the Map paradigm in which each feature instead of being partitioned is sent entirely to all the filters. This process is known as multicast.

A synchronization point is needed in order to compute the final outcome of the stage. The detection window is automatically discarded if the outcome of one of the stages is negative. This composition of patterns can be seen in Fig. 42.

The face detection cascade of classifiers has 22 stages. The number of filters of each stage can be seen in the Tab. 2 below.

Stage	0	1	2	3	4	5	6	7	8	9	10	11	12
Filters	3	15	21	39	33	44	50	51	56	71	80	103	111

Stage	13	14	15	16	17	18	19	20	21
Filters	102	135	137	140	159	177	181	210	213

Table 2 b): Stages of Face Detection Classifier

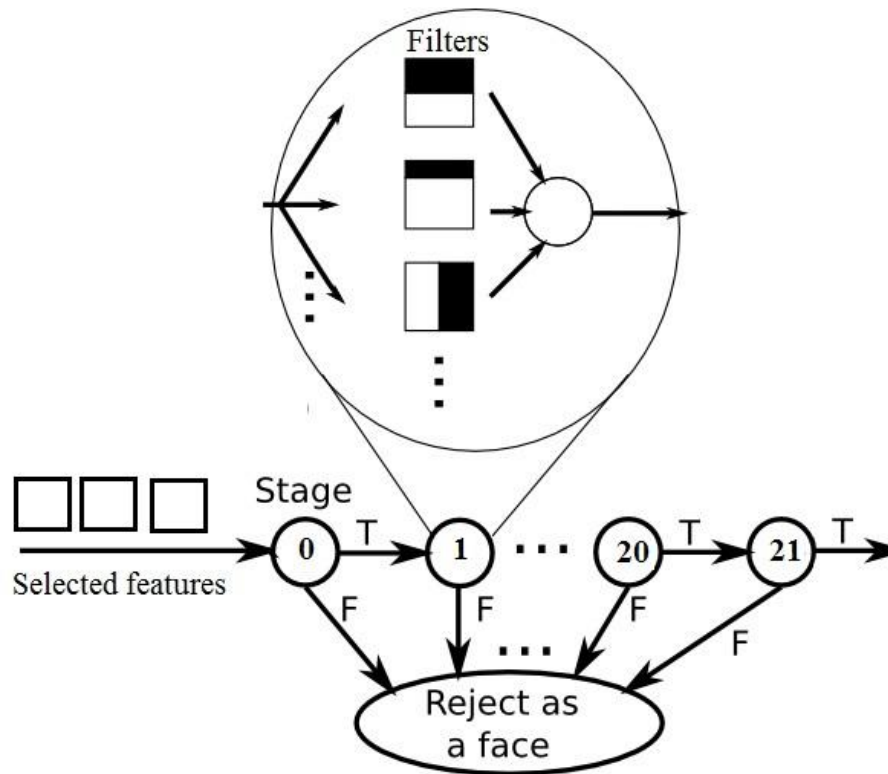


Figure 42: CASCADE OF CLASSIFIERS

**Image Scale:** In order to find all the possible faces with different sizes, we need to scale down the input image according to the specified scale factor until it has the same size of the detection window. Each scale of the input image can be processed in parallel without any need for synchronization. The schema for this case is the same of the figure showed on the Farm paradigm (Fig. 36) in such a way that each scale would be send to a worker.

The overall parallelization scheme can be seen in the Fig. 43 below in which the red circles represent Fig 41.

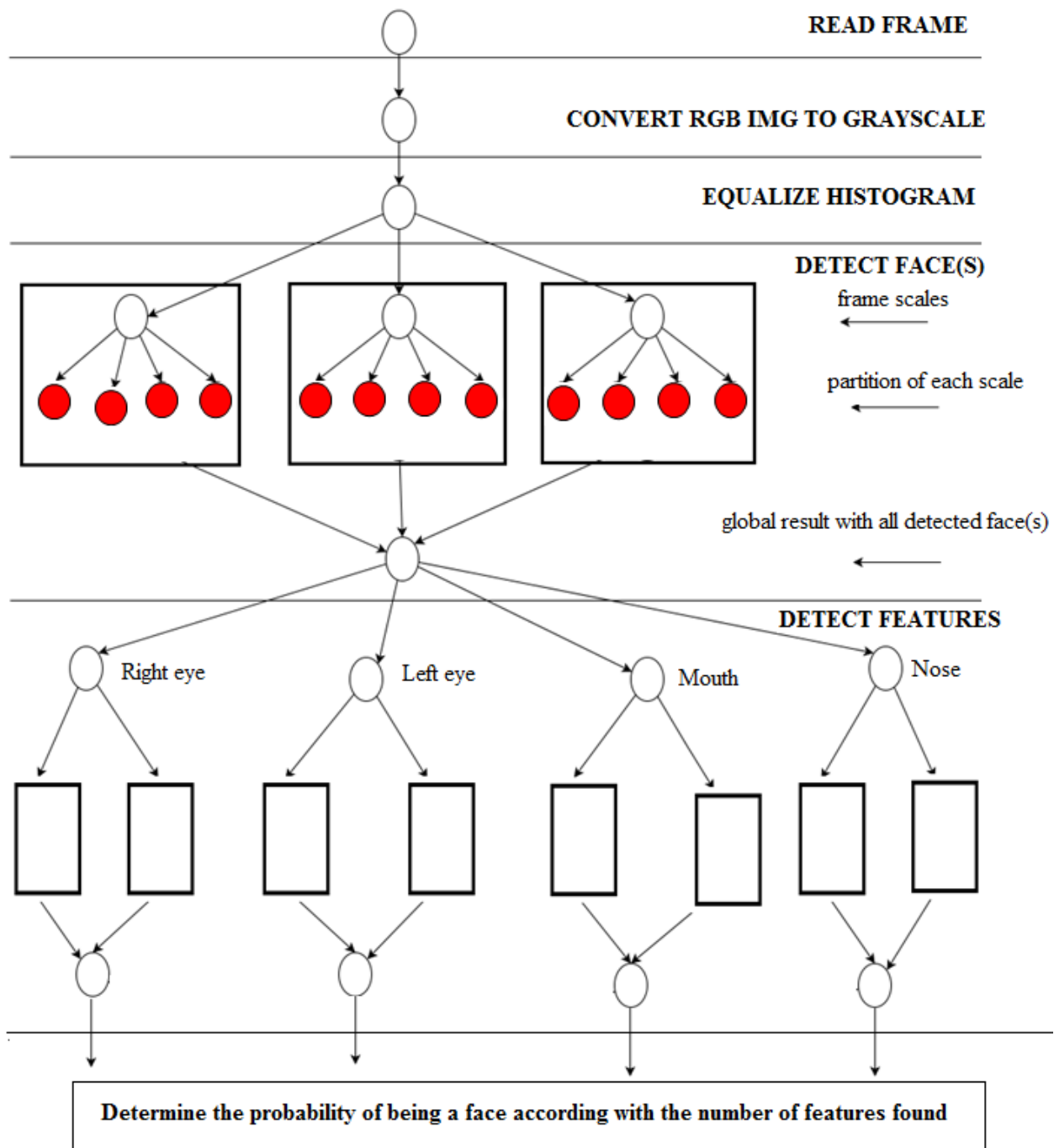


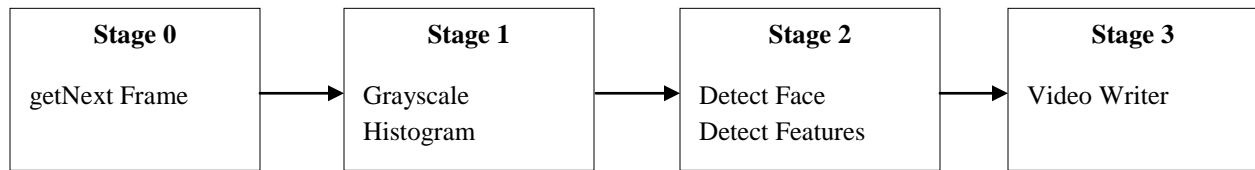
Figure 43: OVERALL PARALLELIZATION SCHEME

## 6.4 Additional Parallelism to the OpenCV Viola-Jones Implementation

In this section, we will focus on providing additional parallelism to the OpenCV implementation of Viola-Jones algorithm. We will also discuss ways of improving the exiting OpenCV parallel implementation.

The most straight forward approach to parallelize a stream computation made up of phases that must be executed serially is to implement a pipeline paradigm. One important aspect to take into account when using the pipeline paradigm is that the stages must be somehow balanced in order to exploit all the benefits of this parallelization.

The proposed pipeline stages are the following ones:



The results that support this pipeline configuration are shown in Section 7.1. In order to implement this paradigm, we will use FastFlow.

The pipeline completion time depends on the slowest stage which in this specific case is stage 2. This stage completion time can be approximated by the face detection task time, so the cost model of the pipeline can be derived as:  $T_c = N \times \text{face\_detection}()$  where  $N$  is the size of the stream.

Therefore, in order to fully exploit the pipeline paradigm and taking into account the completion time of each task, it becomes very important to be able to parallelize the face detection procedure internally.

### Face Detection Task Parallelization

The face detection task processes a pyramid of scales of the same frame. The image pyramid allows the face detection to be scale-invariant, that's it, detecting large and small faces using the same detection window.

There is no dependency between the processing relative to the different scales: they can be executed in parallel without any need for synchronization. It's clear that the computation time is different between the scales due to their different size while applying the same detection window.

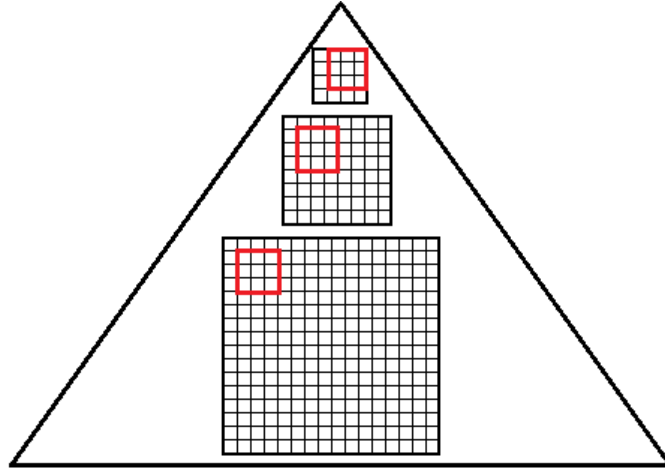


Figure 44: IMAGE PYRAMID

It's important to distinguish two cases:

- a) a more general approach where the goal is to find all the possible faces
- b) a more specific approach where the goal is to find only one face

The face detection optimization specified on Section 5.1.1 should be applied only in case b).

#### 6.4.1 General Approach: Multiple Face Detection

We will start this section by first discussing the OpenCV parallel implementation then propose an improvement to that implementation.

##### Considerations about OpenCV parallel implementation

Although we agree that with a limited number of resources (cores) dividing the computation space into a collection of detection windows is the most valuable parallelization strategy, from the ones defined in Section 6.3, we think that there is still a room for improvement of the current OpenCV parallel implementation.

As seen in the Section 6.2, OpenCV internally parallelizes the computation relative to each scale using the parallel for paradigm. Externally, the scales are executed sequentially, one after the other.

Even though the image scales are of different size, the same number of threads is assigned to compute all of them. This can lead to some inefficiency as it's shown in the Tab. 3 below in which IMG 3 was used for testing. For this specific case, the face detection task generated an image pyramid of 46 scales. We measured the time of each scale using a thread number between 1 and the maximum number of threads available (24 in this case).



Scale	Size	1 thread[ms]	24 threads[ms]
1	2348x2348	624.67	52.55
2	2134x2134	488.82	42.51
3	1940x1940	394.66	31.03
4	1764x1764	1242.92	87.93
5	1604x1604	818.67	59.10
6	1458x1458	664.61	40.90
7	1325x1325	543.27	38.0
8	1205x1205	444.73	30.80
9	1095x1095	365.02	29.78
10	996x996	302.06	23.13
11	905x905	250.30	22.51
12	823x823	208.28	15.75
13	748x748	174.09	14.28
14	680x680	143.71	12.55
15	618x618	118.38	9.75

Scale	Size	1 thread[ms]	24 thread[ms]	Scale	Size	1 thread[ms]	24 threads[ms]
16	562x562	96.93	9.25	31	135x135	4.06	2.51
17	511x511	79.94	8.24	32	122x122	3.2	2.17
18	465x465	65.75	6.64	33	111x111	2.8	2.22
19	422x422	53.72	5.43	34	101x101	1.7	2.02
20	384x384	44.62	4.43	35	92x92	1.53	1.65
21	349x349	36.49	4.25	36	84x84	1.15	4.44
22	317x317	30.46	3.34	37	76x76	1.04	1.62
23	288x288	25.63	2.60	38	69x69	0.97	1.27
24	262x262	20.62	3.16	39	63x63	0.87	1.18
25	238x238	17.51	3.16	40	57x57	0.99	1.04
26	217x217	14.77	3.24	41	52x52	0.76	1.15
27	197x197	10.63	3.10	42	47x47	0.80	0.90
28	179x179	8.64	3.01	43	43x43	0.84	0.95
29	163x163	7.59	2.50	44	39x39	0.76	0.51
30	148x148	5.52	2.46	45	35x35	0.18	0.17
				46	32x32	0.0077	0.007

Table 3: time of each scale using a 1 and 24 threads

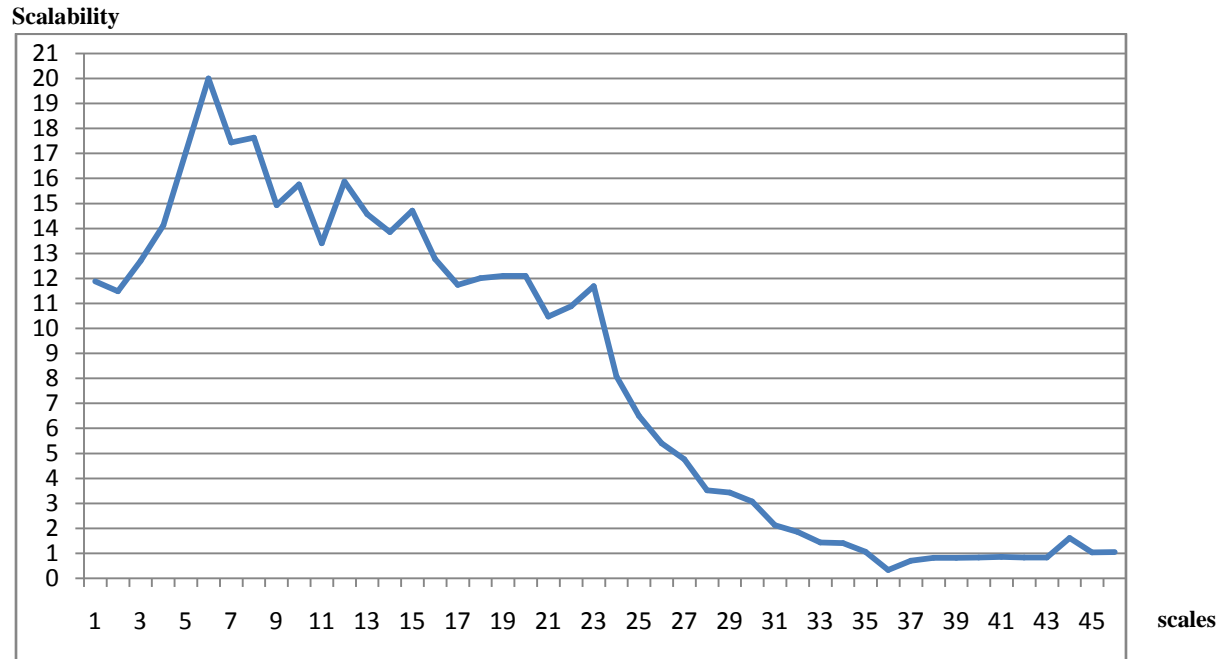


Figure 45: SCALABILITY ACHIEVED WITH 24 THREADS

As evidenced in Fig. 45, the scalability achieved with 24 threads is very significant for scales of bigger size while on the other hand is negligible for scales of smaller size.

If we consider the efficiency, it decreases as the number of threads increases (Fig 46 (a)). When considering the efficiency of each scale with 24 threads (Fig 46 (b)), it decreases drastically from the second half (from scale 24 to 46 of Fig 46 (b)).

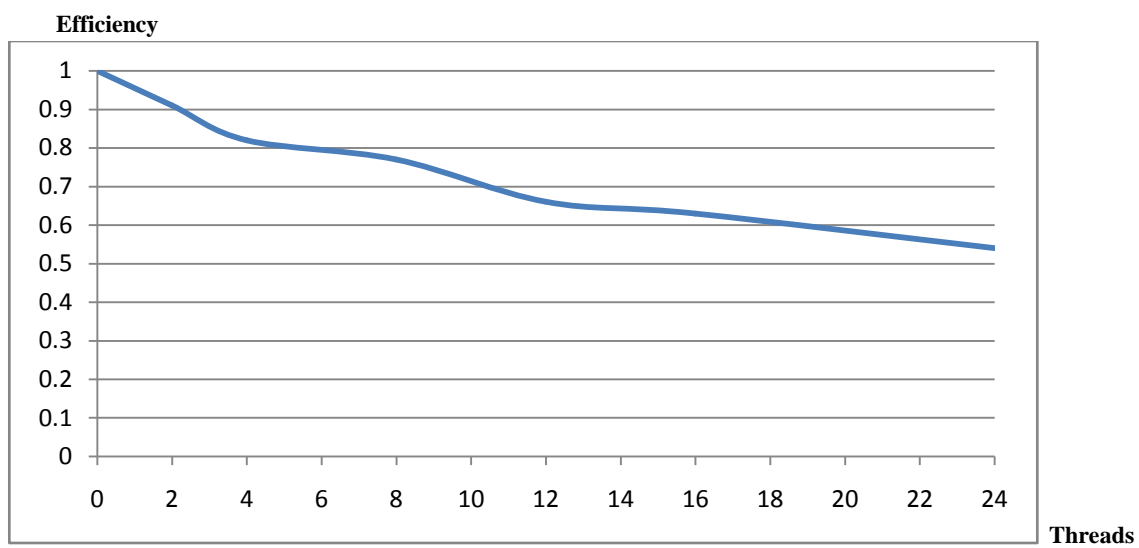


Figure 46 (a): EFFICIENCY ACHIEVED

## Efficiency

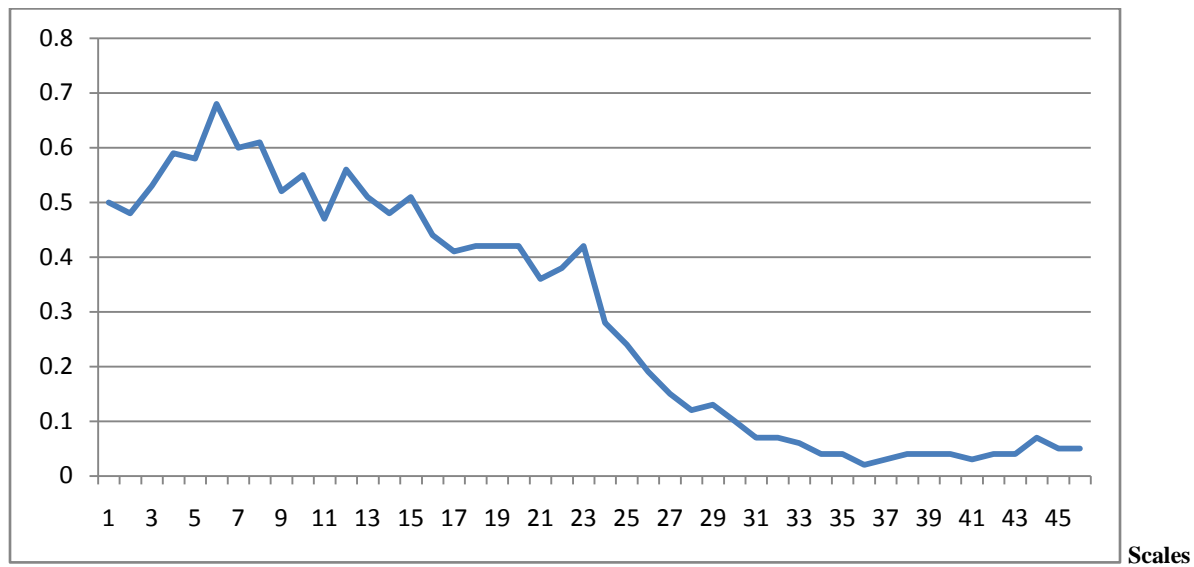
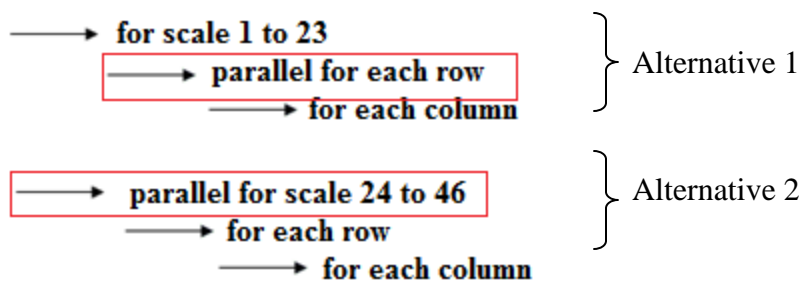


Figure 46 (b): EFFICIENCY ACHIEVED

## Improvement to the OpenCV parallel implementation

Analyzing deeply the results obtained from the Tab. 3 above, it's possible to realize that we can optimize the parallelization with 24 threads in the following alternative ways:



Since in both alternatives we are parallelizing for loops, OpenMP is the most suitable parallelization tool.

We can see that executing in parallel from scale 24 to 46 but inside executing serially each scale, without taking into account any overhead, the computation time can be roughly approximated to the time spent to execute the slowest scale (which is scale 24) with 20.62 milliseconds.

But if we take the OpenCV approach which executes the scales serially while performing a row wise parallel computation inside each scale, the execution time from scale 24 to 46 with 24 threads is roughly the sum of the times of each scale on this range which is 44.437 milliseconds.

So with the optimized approach we are able to get about 2.15x speed up with respect to the OpenCV approach.

Assigning computation of different scales to different threads would fall under the category of embarrassingly parallel or farm paradigm. A dynamic load balance technique can be applied in this case such that when each scale finishes its execution, it frees the resource. This technique may achieve better load balancing in all those cases where the exact amount of work of activities could not precisely be defined before the activities actually take place [17].

Based also on other performed tests which demonstrated similar (scalability) results as the ones shown on Fig. 45, a heuristic solution can be proposed such that we divide the total number of scales by half. The first half containing the heavy sized scales would be parallelized using a row wise parallel for, while the second half containing the light sized scales would be parallelized computing in parallel each scale.

From the Fig. 47 is possible to notice that the face detection parallelization is rather coarse-grained as the best scalability was achieved with the largest image size while the smaller one achieved the worst result.

#### Scalability

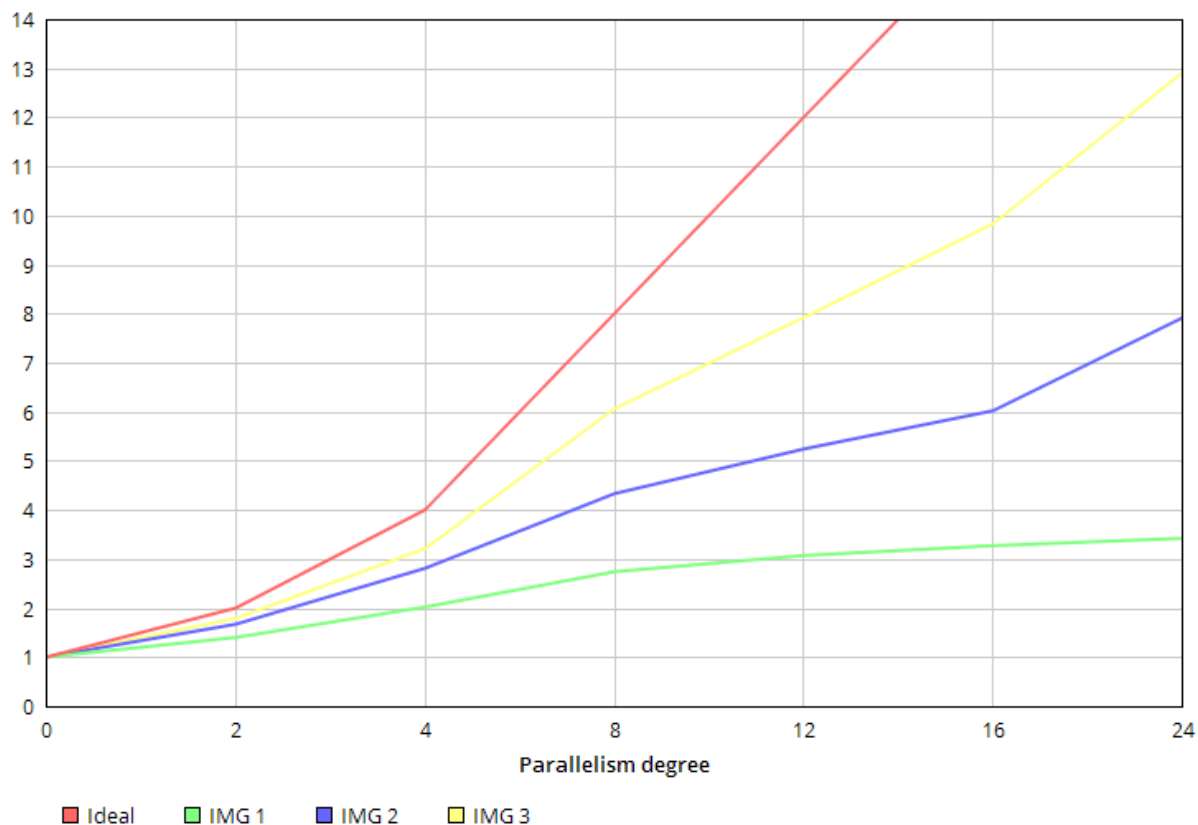


Figure 47: FACE DETECTION PARALLELIZATION

### 6.4.2 Specific Approach: Single Face Detection

For this specific case, before finding a face the parallelization technique is the same as the one described above on the proposed heuristic solution.

Once the face is found, taking advantage of the optimization introduced on Section 5.1.1, where the number of scales and the computation time is reduced drastically, as it is possible to see from the Tab. 4 below, the parallelization paradigm that could be applied is the farm where each scale would be assigned to a specific worker.

One could argue that after finding a face no further parallelization is needed taking into account that the number of scales and the computation time of each scale is reduced drastically. For our specific case, we can see from the Tab. 4 that the sequential implementation would take as much as 4.01 milliseconds to complete while in the best case scenario, without overheads, the parallel implementation would take 0.95 milliseconds.

It is worth to remember that the results of the Tab. 4 were obtained with the largest image of our experiments (IMG 3 with size 2448 x 2448) which represents the worst case of the face sizes that ideally we are looking for.

When introducing further parallelization to this part, we need to take into account the trade-off between the gains of the parallelization and the additional overheads that it comes with as even in the best case scenario the gains can be very negligible (~3 ms for our case).

Scale	Size	1 thread[ms]
1	2348x2348	0.87
2	2134x2134	0.95
3	1940x1940	0.80
4	1764x1764	0.76
5	1604x1604	0.44
6	1458x1458	0.18
7	1325x1325	0.01

Table 4: Optimized results for single face detection

### 6.4.3 Feature Detection Parallelization

In this case we are not interested in parallelizing each feature detection phase internally, as the optimization proposed in Section 5.1.2 already allows achieving higher speed up. The important point is to run all the features (eyes, mouth and nose) in parallel using the MISD pattern which is a variation of the Map paradigm where each worker computes a different code.

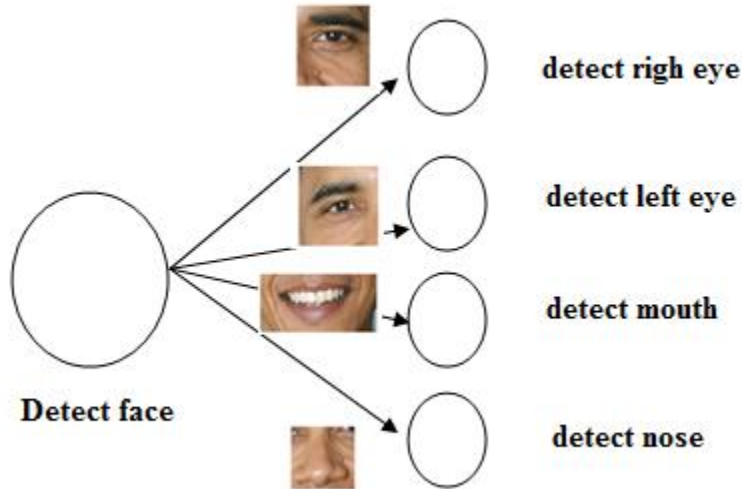


Figure 48: FEATURE DETECTION PARALLELIZATION

A possible approach for detecting features in parallel for multiple faces could be applying the Farm paradigm in which each face would be assigned to a certain worker and inside each worker the MISD paradigm would compute the features in parallel (see Fig 49).

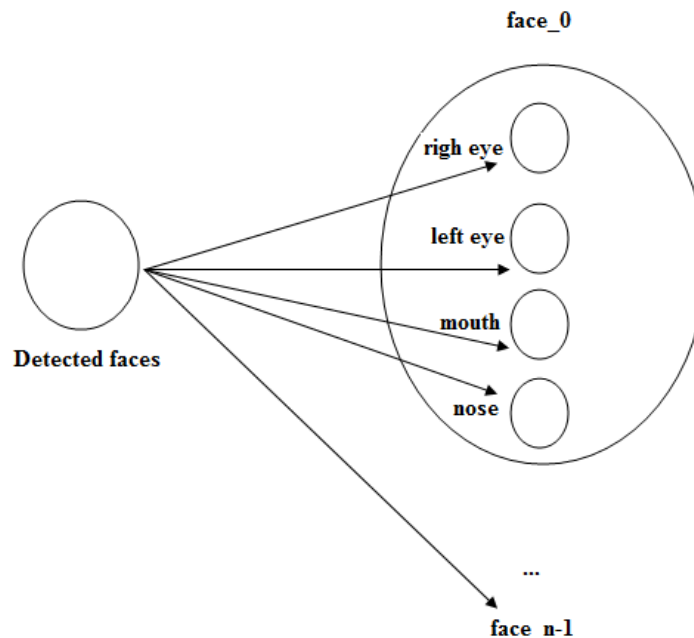


Figure 49: DETECTING FEATURES IN PARALLEL FOR MULTIPLE FACES

## 6.5 Parallel Implementation with GPUs

GPU is a General Purpose Graphic Processing Unit which provides a huge number of cores organized in such a way that they can execute efficiently SIMD code (Single Instruction Multiple Data), in particular multi-threaded code with threads operating on different data with the same code [17].

The GPUs are normally used as co-processors by the CPU in such a way that the input data is sent from CPUs to the GPUs. After processing the results, eventually they are sent back to the CPU main memory. In general, GPUs allows achieving impressive performances in data parallel computations. Their strength is also their weakness as it may perform incredible well only data parallel kind of computations.

The GPU architecture is composed by a large collection of SMs (Stream Multiprocessors) which is a generalization of SIMD architecture to include hardware multithreading [16]. This kind of architecture is known as SIMT (Single Instruction Multiple Thread).

A SM is composed of independent cores (Execution Unit) where each EU executes one hardware thread. A group of 32 threads is called “Warp”. All the threads in a Warp execute together. In other words, each warp is executed in a SIMD fashion (i.e. all threads within it must execute the same instruction at any given time). A warp is the basic scheduling and executing unit in the GPUs.

Computer vision is one of the tasks that often naturally map to GPUs as computer vision solves the “inverse” of the computer graphics problem. While graphics transforms a scene or object description to pixels, vision transforms pixels to higher-level information [26].

### 6.5.1 GPU Parallelization with CUDA

CUDA (Compute Unified Device Architecture) is a C-based programming model proposed by NVIDIA that exposes the parallel computing capabilities of the GPU to application developers in an easy to use manner. It exposes a fast shared memory region that can be accessed by blocks of threads, allows for scattered reads and writes, and optimizes data transfers to and from the GPU [21].

The CUDA programming model offers the GPU as a data-parallel co-processor to the CPU. In the CUDA context the GPU is called device, whereas the CPU is called host. Kernels refer to an application that is executed on GPU. It is important to notice that a CUDA kernel is launched on the GPU as a “grid” of thread blocks (see Fig 50).

Once a kernel is launched, its dimensions cannot change. All blocks in a grid have the same dimension. The total size of a block is limited to 512 threads.

A CUDA program executes sequences of kernels, functions that run under the Single Instruction, Multiple Threads (SIMT) model.

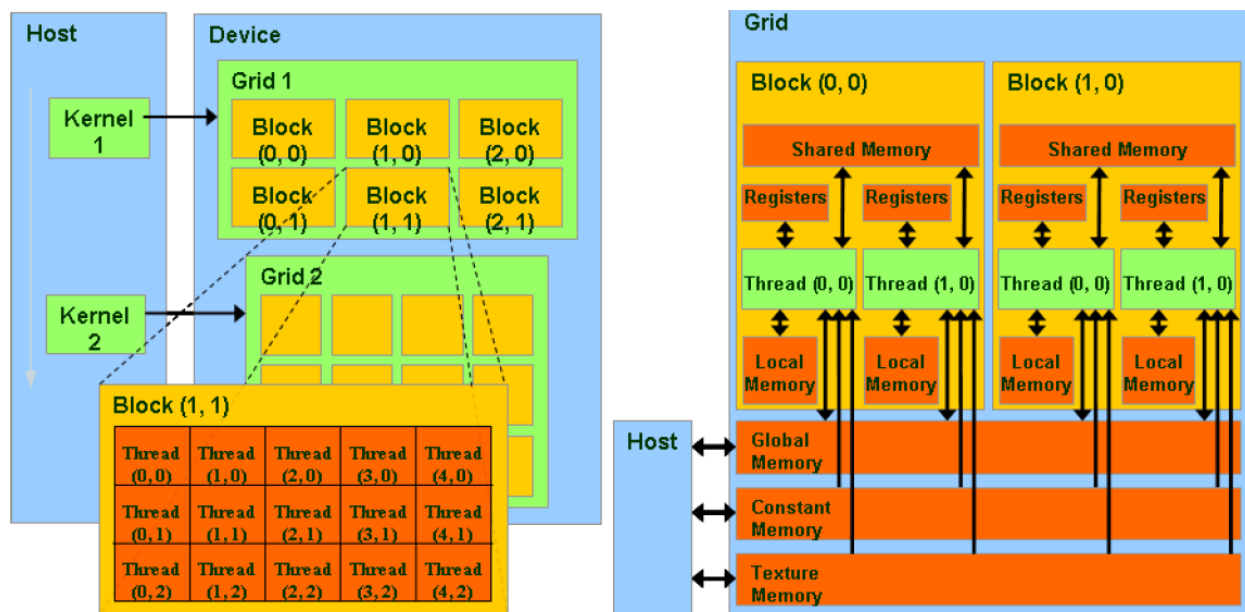


Figure 50: CUDA PROGRAMMING AND MEMORY MODEL. KERNEL 1 IS A TWO DIMENSIONAL 3×2 GRID WITH TWO DIMENSIONAL 5×3 THREAD BLOCKS.

A thread block contains a fixed number of threads and it can span in one (x), two (x, y) or three (x, y, z) dimensions. A grid can span in one or two dimensions. Threads are uniquely identified based on their block index and thread index within the block. All threads in a grid execute the same kernel function.

The threads in a thread block share a local store (usually in the range of 16KB) known as shared memory as well as a 16KB register file. Threads may access a global device memory, which is un-cached and has high access latency, on the order of 20 times higher than that of shared memory or registers [22].

The implementation of the Face Detection with GPU can be designed as sketched in Fig 51.



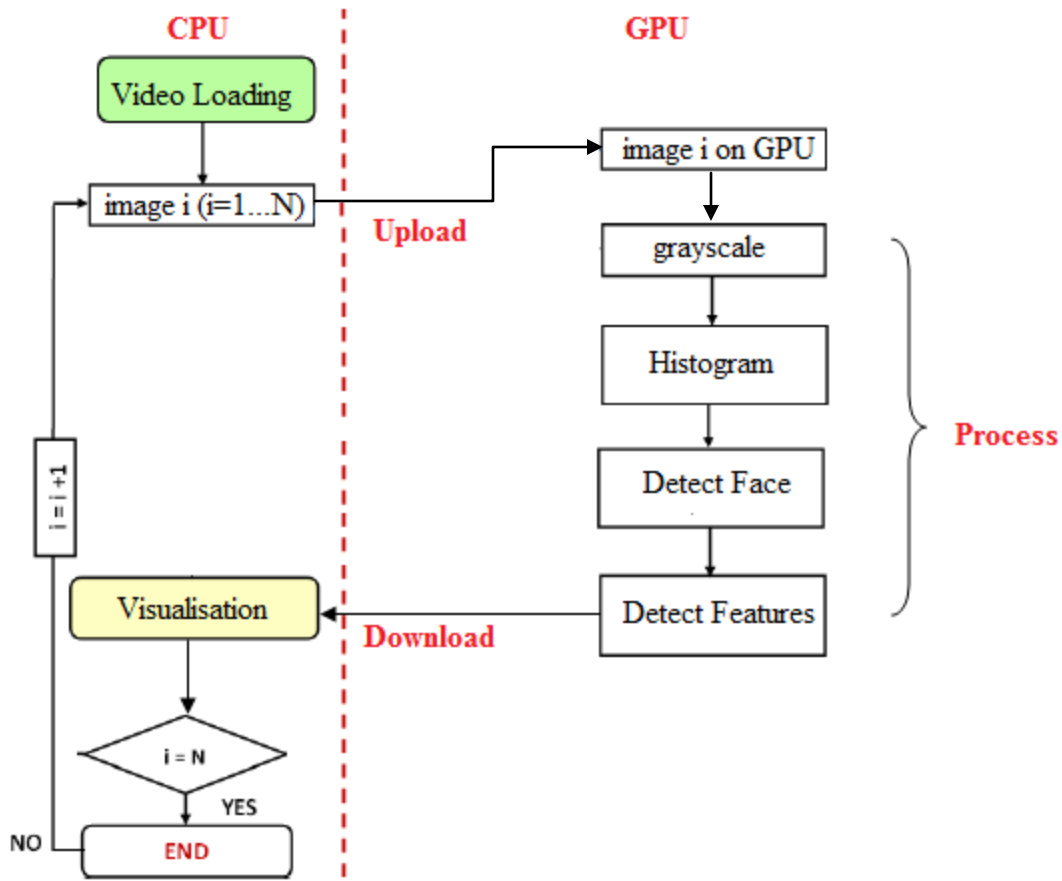


Figure 51: FACE DETECTION WITH GPU

In order to take full advantage of the GPU hardware, [22] specifies some principles must be adhered such as:

- First, it is important to minimize accesses to device memory, by using registers and shared memory to store frequently accessed data
- Second, the transfers to and from the GPU must be overlapped with computation.
- Third, memory accesses to the GPU global and shared memory must avoid bank conflicts using coalesced accesses (technique which allows combining multiple memory accesses into a single transaction [23])
- Fourth, branching should be reduced to a minimum within kernels since threads taking different branches cannot be executed in parallel by a SIMT processor.

Is possible to overlap the operations with communications if we take the approach such as the one specified in the Fig 52.

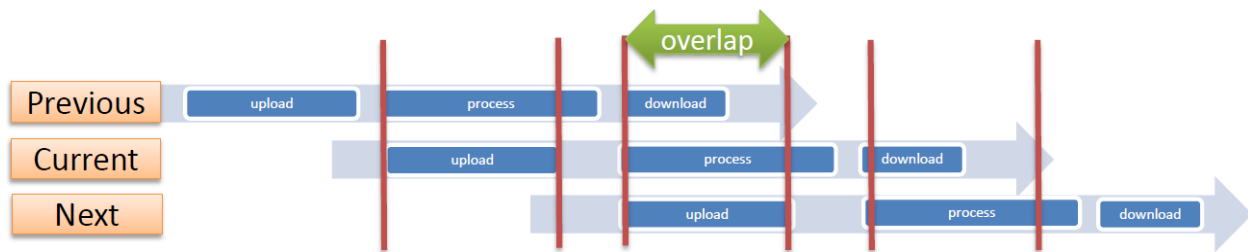
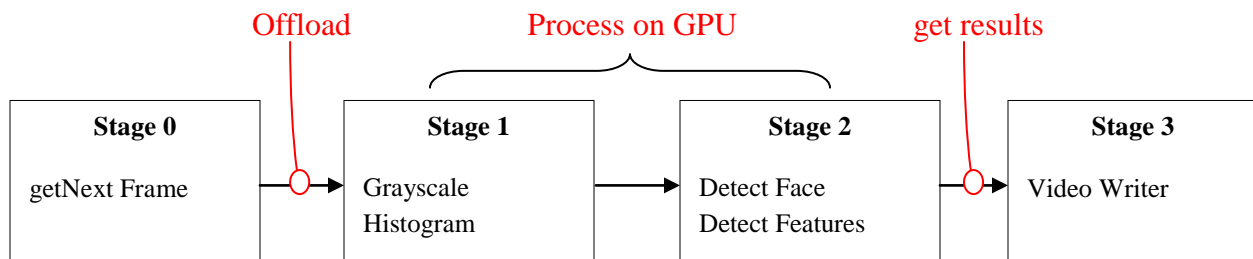


Figure 52: OVERLAP THE OPERATIONS WITH COMMUNICATIONS

So the final pipeline for the GPU version can be seen as:



For the first and third principle mentioned above, for moment being we are relying in the OpenCV CUDA implementation.

## 6.6 Unbalanced Computation of Viola-Jones on Multi-Core Platforms

The Viola-Jones algorithm is based on a cascade of stages. This approach is optimized for sequential implementation because it allows rejecting the majority of the sub-windows on early stages.

This characteristic may not allow the multi-core platforms to be exploited to its full limits because of the unbalance in processing time. The processing time of an image depends on two main factors which are: the processing time of each sub-window and the number of sub-windows to process.

The processing time of a sub-window can vary dramatically according to the complexity of its content. For example, a white image will always take less time to process since no sub-window should be capable of passing the first stage of the cascade. As opposite, a complex image will take much more time to process.

The number of sub-windows to process depends on the size of sub-window itself, the size of the image and the number of scales generated.

So, the unbalanced and the unpredictable processing time of the Viola-Jones algorithm may prevent the multi-core platforms from achieving maximum speed-ups.

### 6.6.1 The GPU Case

In this section, we will address the unbalanced computation problem on GPUs. Several authors were consulted but our main source of reference is [35], as in our opinion, they are the ones that best addressed this problem.

Despite the high computational throughput, the GPU doesn't perform at his best for face detection algorithm because of the irregular workloads. Traditionally, GPU achieves better performance for data level parallelism tasks with regular input and output.

As discussed in section 3.3, Viola-Jones algorithm uses a cascade of classifiers to detect faces. This algorithm uses simpler classifiers to reject the majority of image before more complex classifiers are called. When a window is labeled as "not a face", it is immediately discarded.

As shown in Fig. 53, the Viola-Jones algorithm can result in serious unbalanced computation when implemented on GPUs.

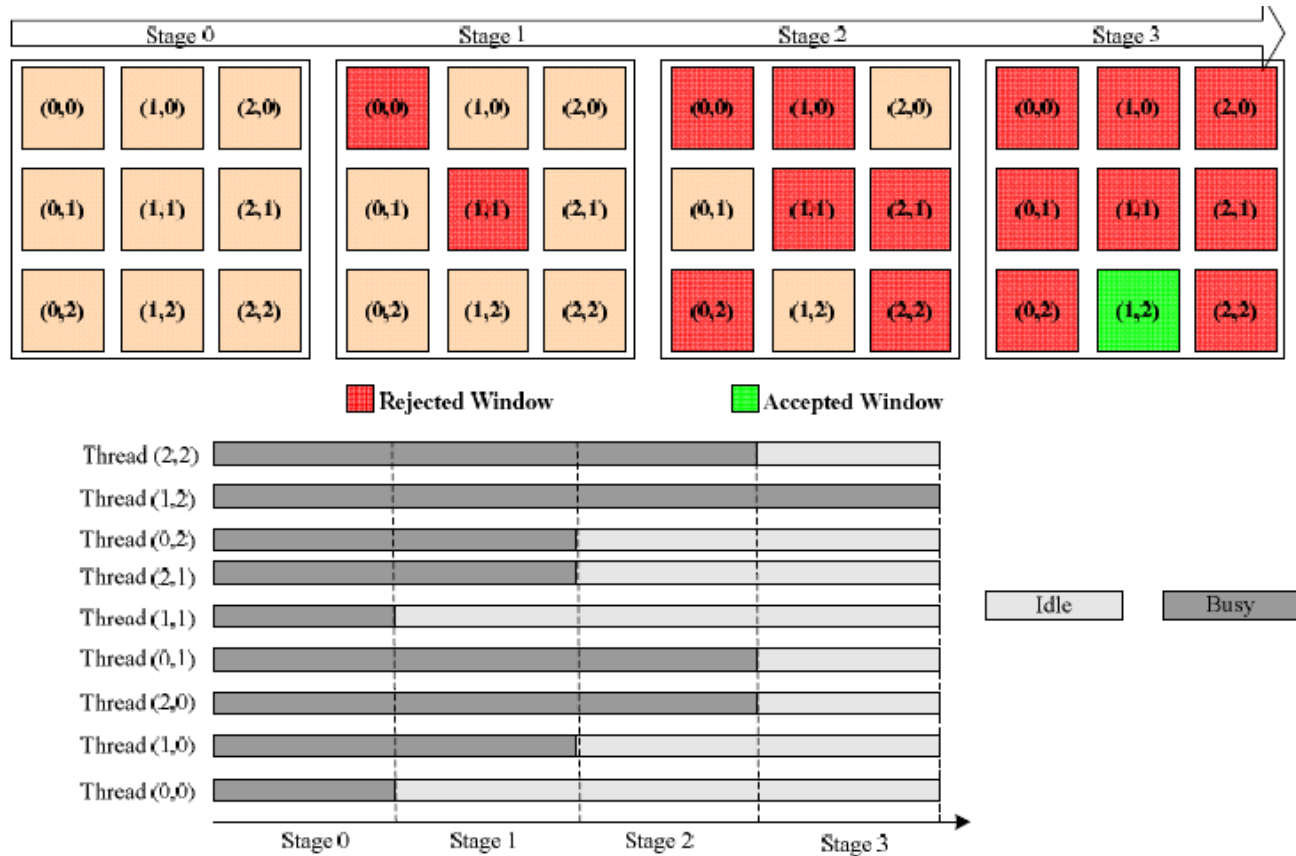


Figure 53: VIOLA-JONES ALGORITHM UNBALANCED COMPUTATION WHEN IMPLEMENTED ON GPUS

The Fig. 53 shows the detect progress using a cascade of classifier assuming that there are 9 detection windows and a thread block contains 9 threads in such a way that each thread works on exactly one detection window.

After four stage detection, only one window is accepted as a face. At the first stage, all 9 threads process their respective window in parallel, and there are no idle threads. However, after stage 0 processing, two windows are labeled as “not a face” and are discarded. So at stage 1, thread (0,0) and (1,1) are in idle state, while other threads continue running.

More windows are discarded after the processing of stage 1. As shown in Fig.53, at stage 2, only three threads are working while the remaining are in idle state. At the final stage, only one thread is working while the others are idle. The final result is that, there is serious unbalanced computation among those 9 threads. More precisely, we have three level of unbalance on Viola-Jones algorithm for GPUs.

**Thread Level unbalance:** the GPUs make use of the SIMD programming model, the unit of execution and scheduling is a warp. For the face detection case, one thread works on one sub-window, the running time of a warp is determined by the thread which has the longest running time. As described in the Fig. 53, there is a serious load unbalance among threads within a warp, which can lead to significant performance decrease.

**Block Level unbalance:** the input image is divided into blocks. In the cases that, only a few blocks successfully pass the last stage, in order to identify a face while the remaining majority of them are quickly idle as a result of the early stages, we will also have unbalance among blocks.

**Image Level unbalance:** In order to find all faces with different size, we need to scale down image according to a scale factor. As the image dimensions progressively decrease, fewer and fewer threads are needed. This progress will result in a smaller grid of thread blocks as the scale factor grows. We need to re-launch the kernel for each scale factor. This can result in unbalanced computation when the image is not large enough to provide enough threads.

In order to solve the above mentioned Viola-Jones load unbalance problem on GPUs, [35] proposed an optimized parallel implementation in which five main optimization techniques are introduced such as: Warp Front Size Work Granularity, Persistent Threads, Uberkernel, Local Queues and Global Queues.

These techniques should help algorithms whose memory access and work distribution are irregular by eliminating unbalanced computation in order to improve the performance on GPUs.

**Warp Front Size Work Granularity:** in this technique the number of threads per block is equal to the Warp size (32 threads). This allows removing explicit synchronization barriers. Warp is the basic scheduling and executing unit for GPUs, so we can remove the barrier operations if one block contains only one warp.

**Uberkernel:** this technique tries to avoid wasting of resources when the scaled down images cannot provide enough threads to occupy all compute units. Uberkernel packs multiple different tasks into a single physical kernel, expressing task parallelism within that kernel without overhead of switching between kernels.

**Persistent Threads:** this technique launches only enough threads to fill the GPUs and to hide memory access latency, and leaves those threads alive throughout the entire kernel. Persistent thread is well suited for efficient production and consumption of irregularly-parallel work.

**Local Queues:** there is serious unbalanced computation among threads within a single block because some threads may need to evaluate much more features and stages than other threads. In order to address this problem, one local queue is built for each block in the shared memory. All the detection windows that will be computed by all threads within a block are pushed into the local queue.

**Global Queues:** local queues can provide load balancing within a block. However, there is also serious unbalanced computation among blocks. Global queue are used to dispatch tasks among blocks as even as possible. Nevertheless, global queue cannot eliminate but only reduce unbalanced computation among blocks.

For the moment being we are focused on addressing the load unbalance problem, future work will include study in details the techniques mentioned above in order to solve this problem.

### 6.6.2 The CPU Case

If we consider CPU multi-core platforms, the same load unbalancing is verified. The Fig. 54 shows the detect process using a cascade of classifier assuming that there are 4 detection windows and 4 threads in such a way that each thread works on exactly one detection window.

As it is possible to notice, after the processing of stage 0, thread 1 is in the idle state. After stage 2, also thread 3 and 4 becomes idle.

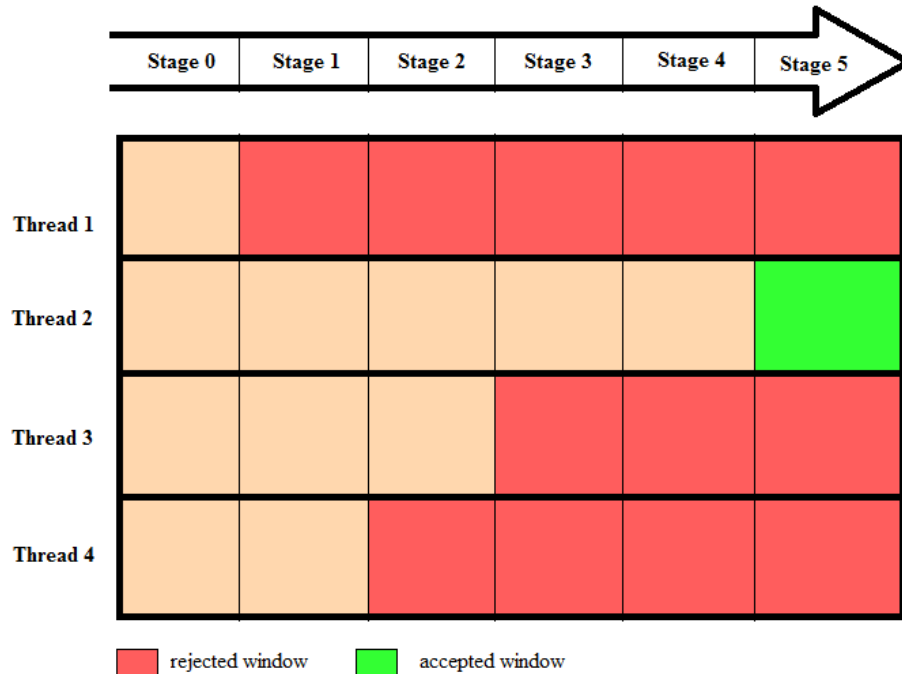


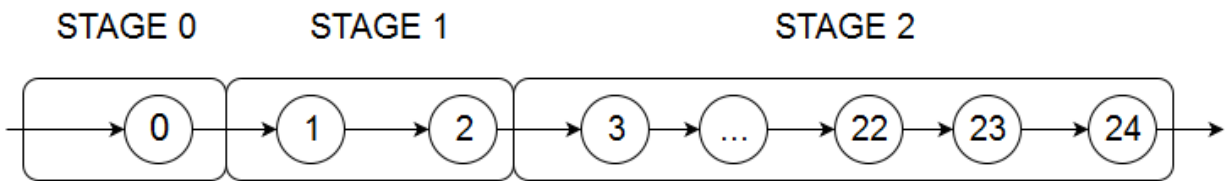
Figure 54: VIOLA-JONES ALGORITHM UNBALANCED COMPUTATION ON CPUS

The processing time of the first stage is constant and independent of the image content. The processing time depends largely on the detection windows evaluated as positive in the previous stages.

The cascade for the face detection has 22 stages as mentioned on Section 6.2. On average, 35% of the memory accessing and processing load takes place in the first two stages while 32% takes place in all the remaining stages combined [28].

In order to try to solve the load unbalance we propose a solution based on two parallelization strategy. First, taking into account the percentage of memory access and processing load mentioned above, we could implement a three stage pipeline.

The first stage of the cascade would be assigned to the first pipeline stage, the second and third stage of the cascade would be assigned to the second stage of the pipeline while the rest of the cascade stages would be assigned to the last stage of the pipeline.



The second parallelization strategy consists in implementing a dynamic technique called job stealing which is in general used at run time and achieve load balancing by dynamically assigning and re-assigning work to the concurrent activities.

When a thread process has finished computing its entire assigned task(s) (e.g a row of the image), instead of becoming idle, it can randomly choose one of the active working thread and steal some task to be computed from those previously assigned to that thread.

The algorithm is the following one: (see Fig. 55)

- 1) When a thread process has finished
- 2) Select a victim
- 3) Steal some task from the victim
- 4) Return to point 1

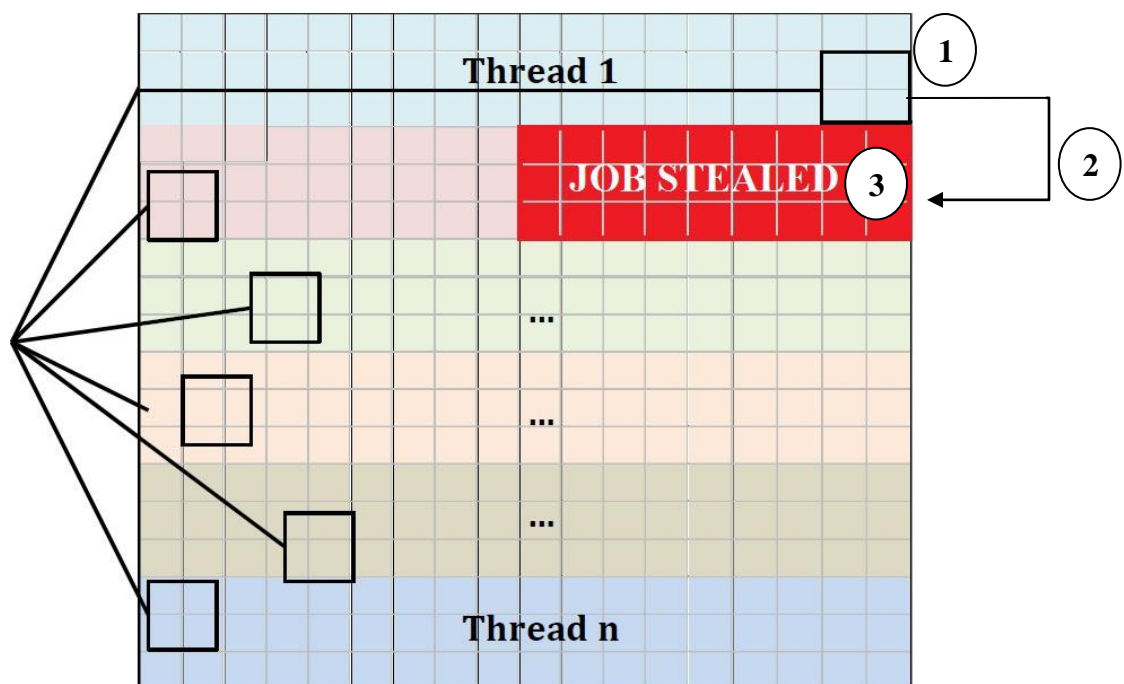


Figure 55: JOB STEALING

When implementing these techniques, it's important to take into account a larger overhead related to the management of dynamic work assignment/re-assignment. So the drawback of this algorithm is the synchronization it introduces to aspects that naturally does not need any synchronization. Also, a good procedure is needed in order to select the victim.

Our final solution can be seen in the Fig. 56 in which each detection window applies the three stage pipeline.

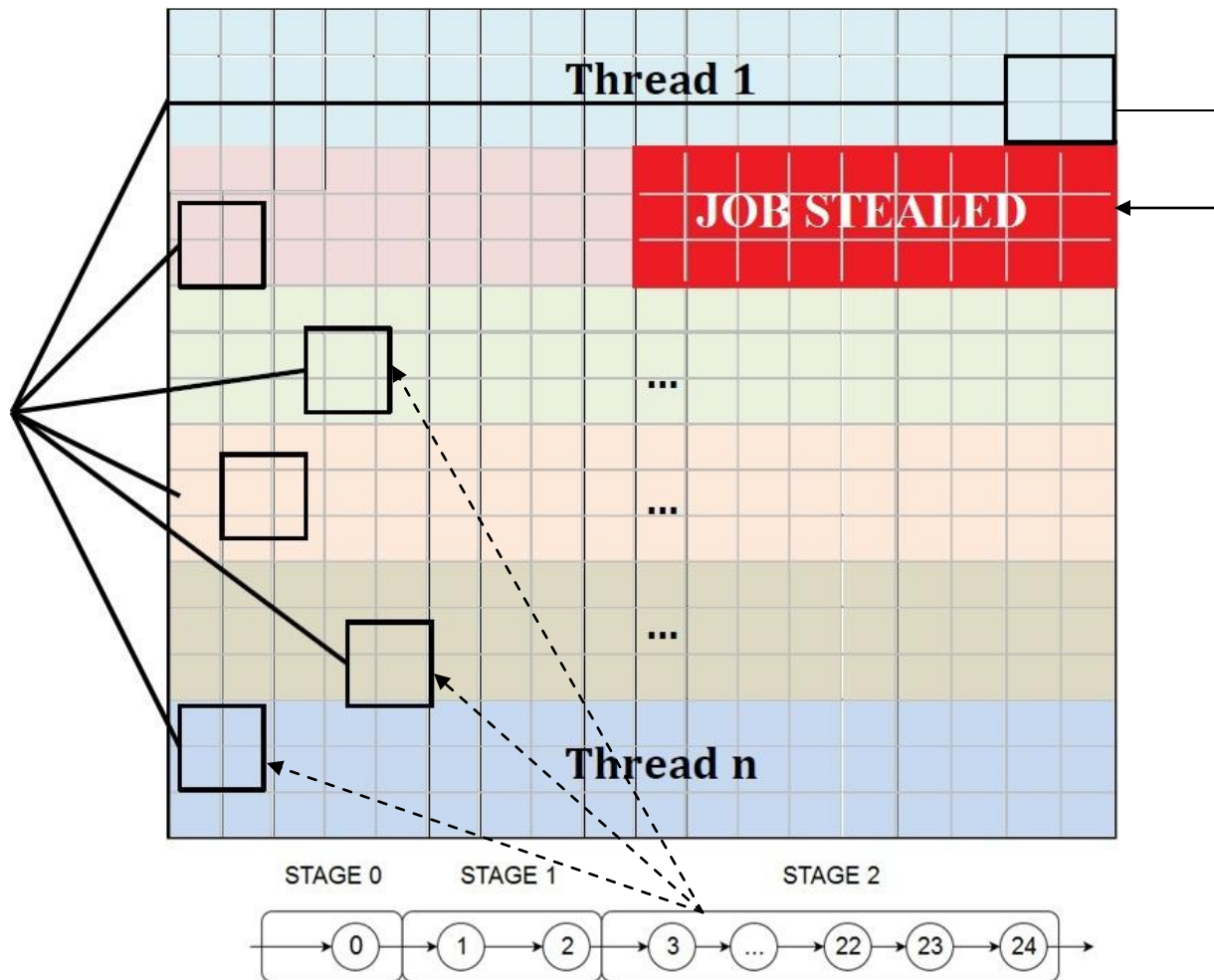


Figure 56: TWO PARALLELIZATION STRATEGIES COMBINED



## 7. Tests

In this section, we will present all the tests performed that supports our implementation. We will start by analyzing the possible configuration of stages for the pipeline paradigm introduced on Section 6.4 followed by the tests for the multiple and single face detection introduced on Sections 6.4.1 and 6.4.2.

Real-Time coverability on video frames is also another set of tests that will be presented followed by the tests that verifies the number of sub-windows processed by the stages of the cascade classifiers.

The last set of tests includes verifying the reliability of our application in scenarios such as distance, light conditions and limitations of the original Viola-Jones algorithm.

All tests were run on an dual 12 core AMD Magny Cours equipped with a NVIDIA Tesla C2050 GPU with 14 Streaming Processors (32 cores each).

When performing the tests we took into account the studies made by [27] which clearly point out that while varying the number of faces within an image can have a very negligible impact on the performance, increasing the resolution causes a drastic performance decline.

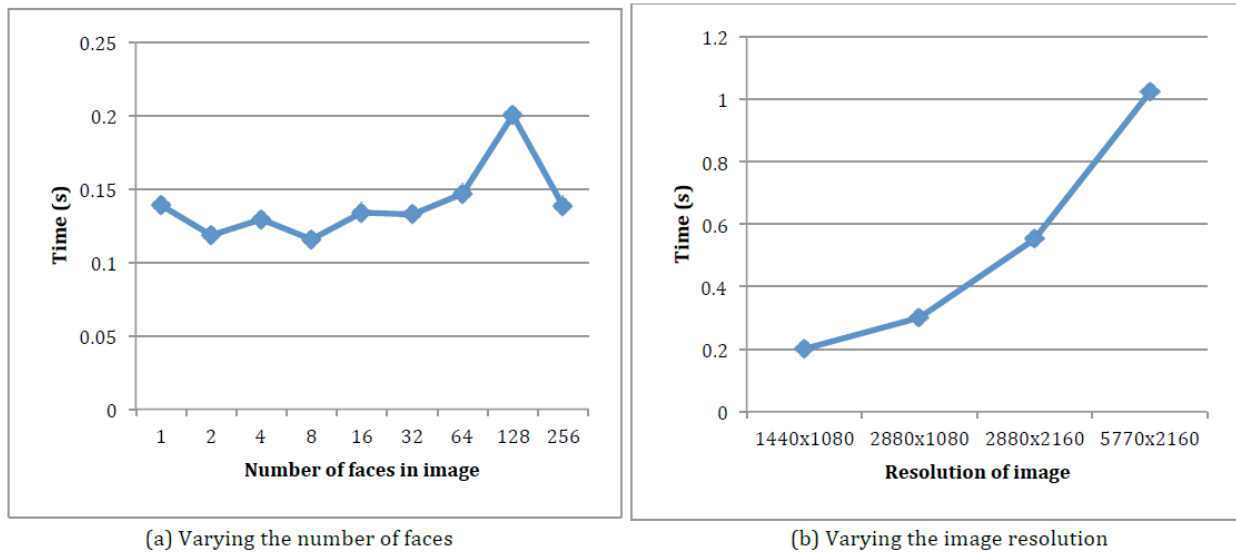


Figure 57: RESULT OF THE STUDIES MADE BY [27]

## 7.1 Pipeline stages

In order to find the best pipeline configuration of stages, we tested two different possibilities for the two most critical tasks namely face and feature detection. For this specific case, our testing set was made of 5 video frames with different lengths.

As it's possible to notice from the Fig. 58 below, there is no performance gain in separating face and features detection in a 5 stage pipeline or putting them together in a 4 stage pipeline. Based on these results, we decided to apply the 4 stage pipeline.

**Time (ms)**

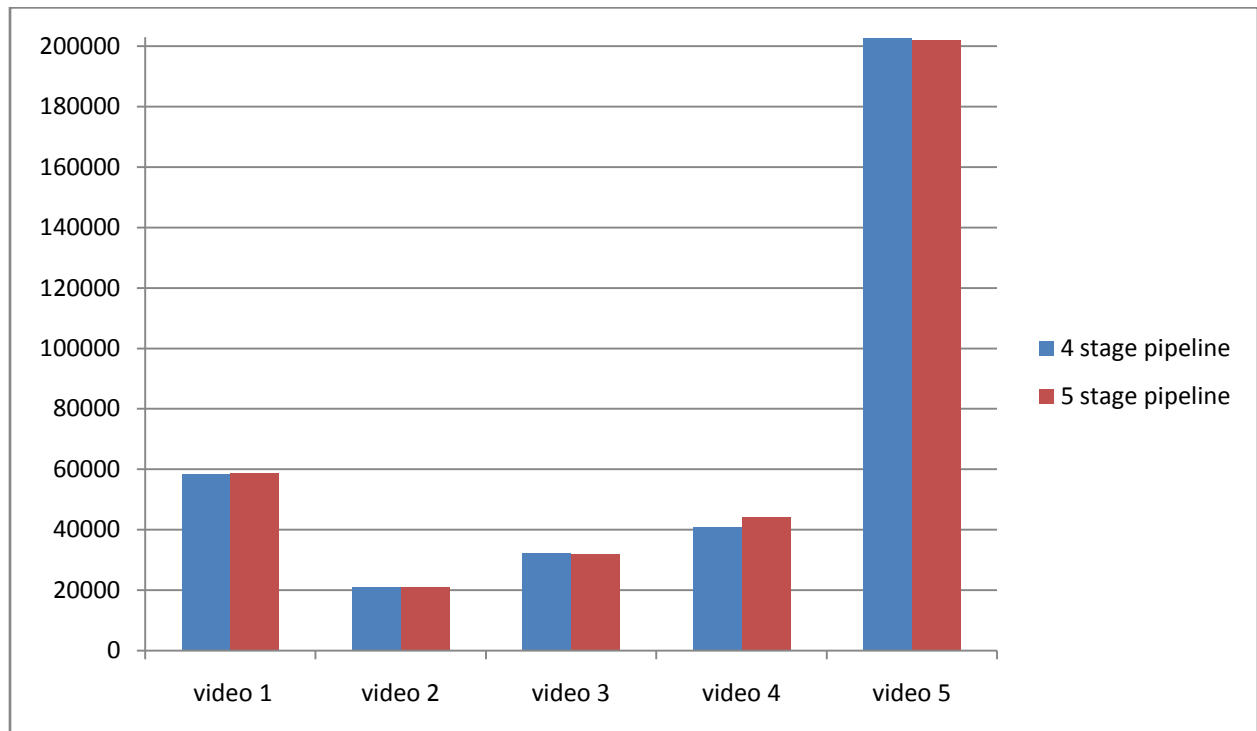


Figure 58: PIPELINE STAGES

## 7.2 Multiple and single Face Detection


In this section our goal was to test the two face detection approaches specified previously in Sections 6.4.1 and 6.4.2.


### 7.2.1 Tests for the general case: Multiple Face Detection


For this case, it was tested the approach suggested on Section 6.4.1, so only the features optimization (described on Section 5.1.2) was applied. Detect features means the global time to detect eyes, mouth and nose.

The parallel CPU version was implemented with the pipeline stages specified on Section 6.4 in which for the task detect face 20 threads were assigned. The GPU implementation was performed based on the pipeline specified on Section 6.5.1.

 Speed up with respect to the sequential version

Size: 444x600 Trials: 200	Task	Sequential Time in ms	Seq optimized Time in ms	CPU Parallel Time in ms		GPU Time in ms	
 IMG 1	gray scale	0.973	0.85	0.839	1.16x	0.322	3.02x
	histogram	2.377	2.17	1.306	1.82x	0.541	4.39x
	d. face	309.673	309.82	96.970	3.16x	21.005	14.72x
	d. features	7.590	5.05	3.008	2.52x	3.701	2.05x
	d. right eye	1.502	1.690	1.879	0.79x	2.717	0.55x
	d. left eye	1.166	1.572	2.135	0.55x	2.151	0.54x
	d. mouth	2.215	1.140	2.477	0.89x	0.841	2.63x
	d. nose	2.707	0.65	1.260	2.14x	1.989	1.36x
	<b>total time</b>	322.324	318.36	102.66	3.14x	26.223	12.29x
	<b>FPS</b>	3.10	3.14	9.74		38.13	

Size: 1000x1094 Trials: 200	Task	Sequential Time in ms	Seq optimized Time in ms	CPU Parallel Time in ms		GPU Time in ms	
 IMG 2	gray scale	3.579	3.481	0.794	4.51x	0.488	7.33x
	histogram	9.242	9.187	1.777	5.20x	2.187	4.23x
	d. face	1712.133	1685.793	279.983	6.12x	46.473	36.84x
	d. features	303.790	13.529	8.398	36.17x	8.591	35.36x
	d. right eye	59.005	4.692	5.486	10.76x	6.455	9.14x
	d. left eye	60.740	4.825	7.193	8.44x	7.708	7.88x
	d. mouth	87.421	1.001	5.948	14.70x	1.041	83.98x
	d. nose	96.622	3.010	2.017	47.90x	2.385	40.51x
	<b>total time</b>	2029.017	1712.269	291.328	6.96x	59.415	34.15x
	<b>FPS</b>	0.49	0.58	3.43		16.83	

Size: 2448x2448 Trials: 200	Task	Sequential Time in ms	Seq optimized Time in ms		CPU Parallel Time in ms		GPU Time in ms	
 IMG 3	gray scale	18.137	18.608		2.987	6.07x	1.026	17.68x
	histogram	45.629	48.078		5.910	7.72x	4.058	11.24x
	d. face	8190.058	8483.401		852.68	9.61x	385.6	22.84x
	d. features	2576.455	29.980	85.94x	17.774	145.0x	17.28	149.07x
	d. right eye	701.246	12.616	55.58x	12.226	57.36x	16.44	42.65x
	d. left eye	609.910	11.858	51.43x	17.373	35.11x	12.94	47.15x
	d. mouth	533.795	3.797	140.6x	6.736	78.25x	3.233	165.11x
	d. nose	731.499	1.706	452.8x	3.148	232.4x	4.673	156.54x
	total time	10830.858	8580.61	1.26x	880.13	12.31x	415.3	26.08x
	FPS	0.09	0.12		1.14		2.41	

Although it is not specified on the OpenCV documentation, the tasks convert image to gray scale and perform histogram equalization are also parallelized internally. Their parallelization is rather coarse-grained just as the face detection task in which highest speed up was achieved with the largest image size.

The features optimization is also coarse-grained, if we take a look at the sequential optimized version is possible to notice that the speed up achieves as far as 86 with the largest image (IMG 3) and 1.50 with the smallest one (IMG 1). Running all the features in parallel allows a speed up ~ 2x with respect to the sequential optimized version.

The CPU parallelization demonstrated better speed up with the IMG 3 (12.31x) with respect to IMG 2 (~ 7x) and IMG 1 (~ 3x) while the GPU parallelization showed a better speed up with the intermediate size image (34.15x) with respect to IMG 1 (12.29x) and IMG 3 (26.06x).

The IMG 1 was able to achieve real-time processing with GPU (~38 FPS) and a quasi-real-time with the CPU parallelized version (~ 10 FPS). IMG 2 was able to achieve a quasi-real-time processing with GPU (~ 17 FPS).

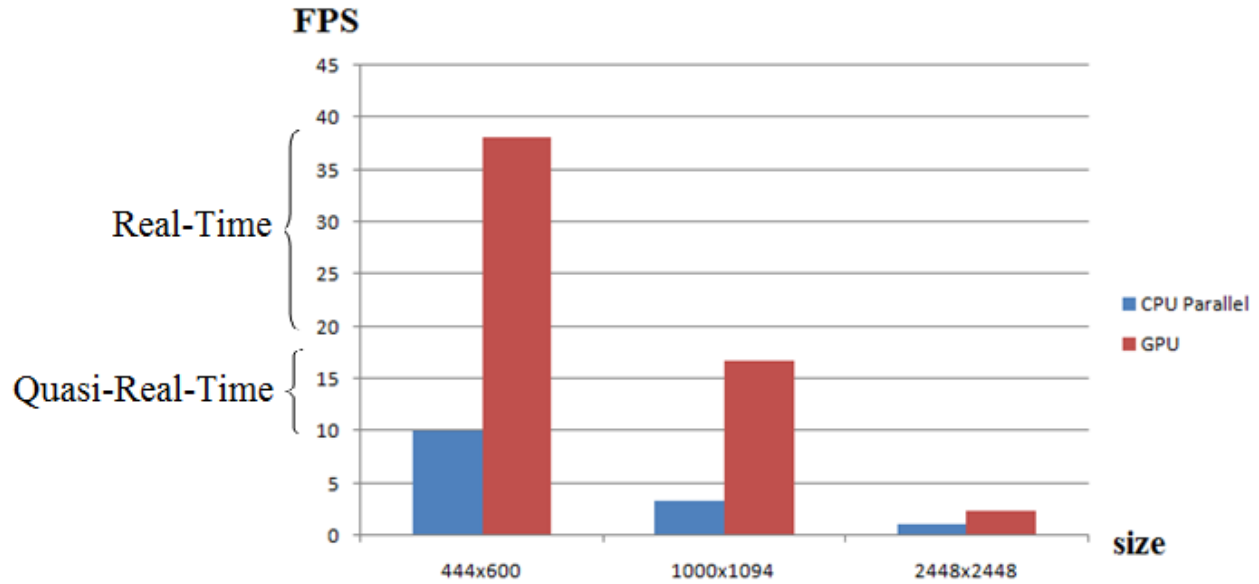





Figure 59: Multiple Face Detection

### 7.2.2 Tests for the specific case: Single Face Detection

For this case, it was tested the approach suggested on Section 6.4.2, so both features (Section 5.1.2) and face detection (Section 5.1.1) optimizations were applied. For the sake of comparison with the general case, exactly the same images were used for testing.

Size: 444x600 Trials: 200	Task	Sequential Time in ms	Seq optimized Time in ms		CPU Parallel Time in ms		GPU Time in ms	
 IMG 1	gray scale	0.973	0.821		0.568	1.71x	0.319	3.05x
	histogram	2.377	2.146		1.469	1.62x	0.548	4.33x
	d. face	309.673	6.032	51.33x	5.191	59.66x	6.631	46.70x
	d. features	7.590	6.892	1.10x	4.301	1.76x	3.318	2.29x
	d. right eye	1.502	2.036	0.73x	2.927	0.51x	2.576	0.58x
	d. left eye	1.166	1.921	0.60x	3.418	0.34x	2.936	0.39x
	d. mouth	2.215	1.707	1.29x	3.557	0.62x	0.817	2.71x
	d. nose	2.707	1.228	2.21x	2.500	1.08x	1.987	1.36x
	total time	322.324	16.507	19.52x	11.875	27.14x	11.491	28.05x
	FPS	3.10	60.58		84.21		87.02	

Size: 1000x1094 Trials: 200	Task	Sequential Time in ms	Seq optimized Time in ms		CPU Parallel Time in ms		GPU Time in ms	
 IMG 2	gray scale	3.579	3.612		0.906	3.95x	0.496	7.21x
	histogram	9.242	9.597		2.244	4.12x	2.197	4.21x
	d. face	1712.133	14.958	114.46x	9.113	187.88x	8.013	213.67x
	d. features	303.790	14.750	20.59x	9.698	31.33x	7.362	41.26x
	d. right eye	59.005	4.790	12.32x	6.145	9.60x	5.354	11.02x
	d. left eye	60.740	5.275	11.51x	8.626	7.04x	6.515	9.32x
	d. mouth	87.421	3.453	25.32x	7.432	11.76x	1.063	82.24x
	d. nose	96.622	1.230	78.55x	2.776	34.81x	1.429	67.62
	total time	2029.017	43.182	49.99x	22.33	90.83x	19.729	102.85x
	FPS	0.49	23.16		44.76		50.69	

Size: 2448x2448 Trials: 200	Task	Sequential Time in ms	Seq optimized Time in ms		CPU Parallel Time in ms		GPU Time in ms	
 IMG 3	gray scale	18.137	18.862		3.017	6.01x	1.027	17.66x
	histogram	45.629	46.983		5.897	7.74x	4.116	11.09x
	d. face	8190.058	47.961	170.93x	11.45	715.16x	16.90	484.36x
	d. features	2576.455	31.976	80.57x	18.63	138.27x	16.00	161.0x
	d. right eye	701.246	13.581	51.63x	14.39	48.71x	15.14	46.32x
	d. left eye	609.910	12.200	49.99x	17.74	34.36x	12.12	50.31x
	d. mouth	533.795	3.948	135.20x	7.289	73.23x	3.136	170.22x
	d. nose	731.499	2.243	326.13x	3.443	212.46x	4.601	159.0x
	total time	10830.858	146.367	74.0x	39.67	272.96x	45.23	239.46x
	FPS	0.09	6.83		25.20		22.11	

IMG 1 and IMG 2 were able to achieve real-time processing with all the implementations apart from the originally sequential one. IMG 3 was able to achieve real-time processing with the parallel CPU and GPU implementations.

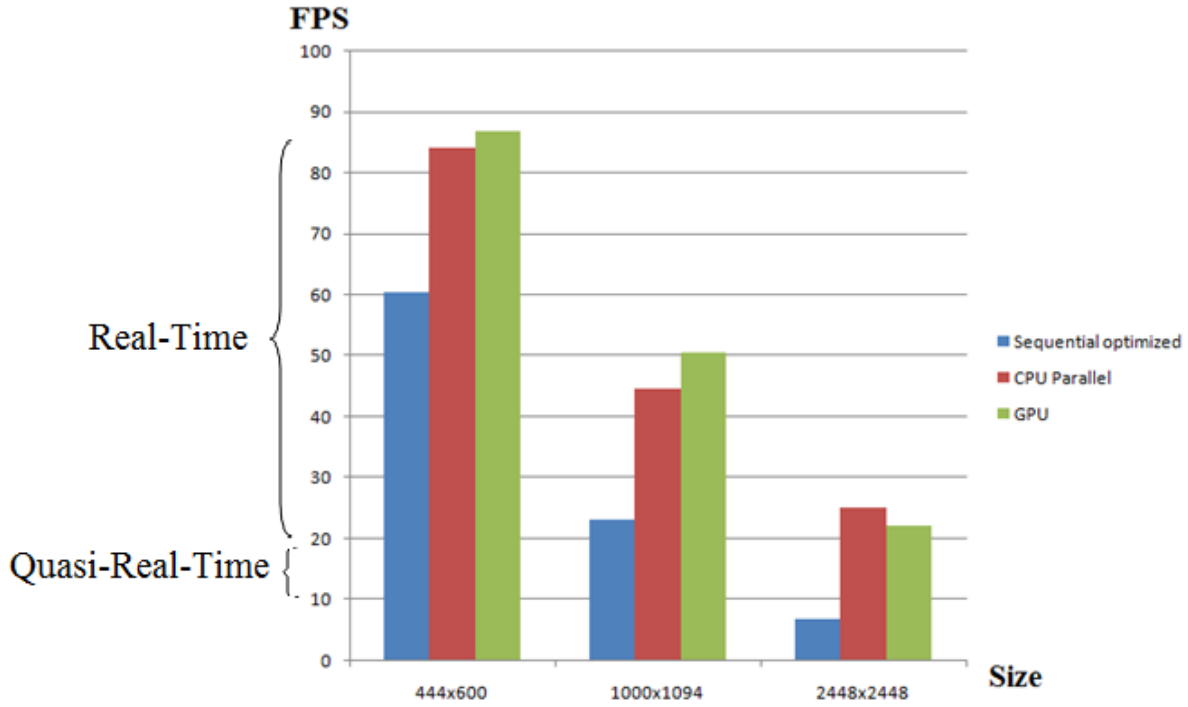


Figure 60: SINGLE FACE DETECTION

### 7.3 Real-Time Coverability on Video Frames

In these tests, we focused on analyzing the FPS at different parts of a video stream. The size of the video frames was 640x480. Only one face was present.

We performed the tests for all the implementations optimized for single face detection (Sequential Optimized, CPU Parallel Optimized, and GPU Optimized) plus the GPU implementation for multiple face detection. The Fig. 61 below shows the implementations mentioned above before finding any face.

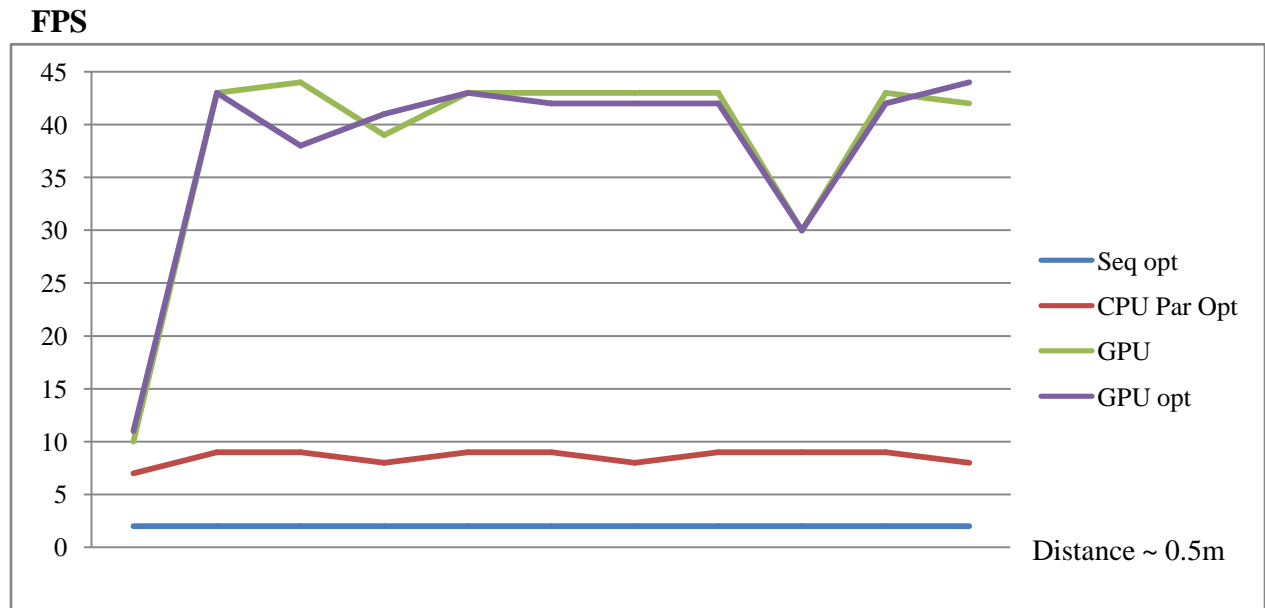


Figure 61: BEFORE FINDING ANY FACE

The optimizations only take place after the face is found, we can see that the sequential optimized implementation runs on a very low FPS while the parallel CPU optimized achieves a quasi-real-time processing. Both GPU implementations were able to achieve a real-time processing.

As is possible to notice from the Fig. 62, after the face is found, all the implementations are able to achieve a real-time processing in which the FPS lies between 20 and 50. It is important to notice that for this case, the face was near to the camera ( $\sim 0.5$  meters).

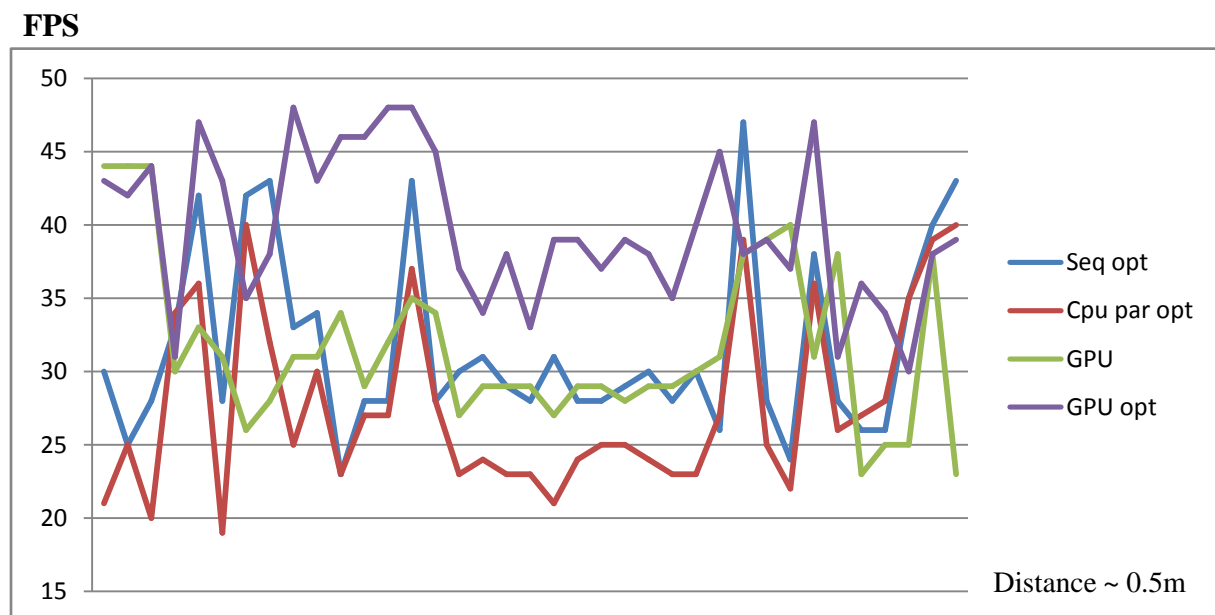


Figure 62: AFTER THE FACE IS FOUND



When the face started to get far away from the camera, until it reached a maximum distance approximated to 4.5 meters, the FPS started to decrease for the sequential optimized version which at the worst case was not able to achieve a quasi-real-time processing while the CPU parallel optimized version achieved a quasi-real-time processing for the worst case.

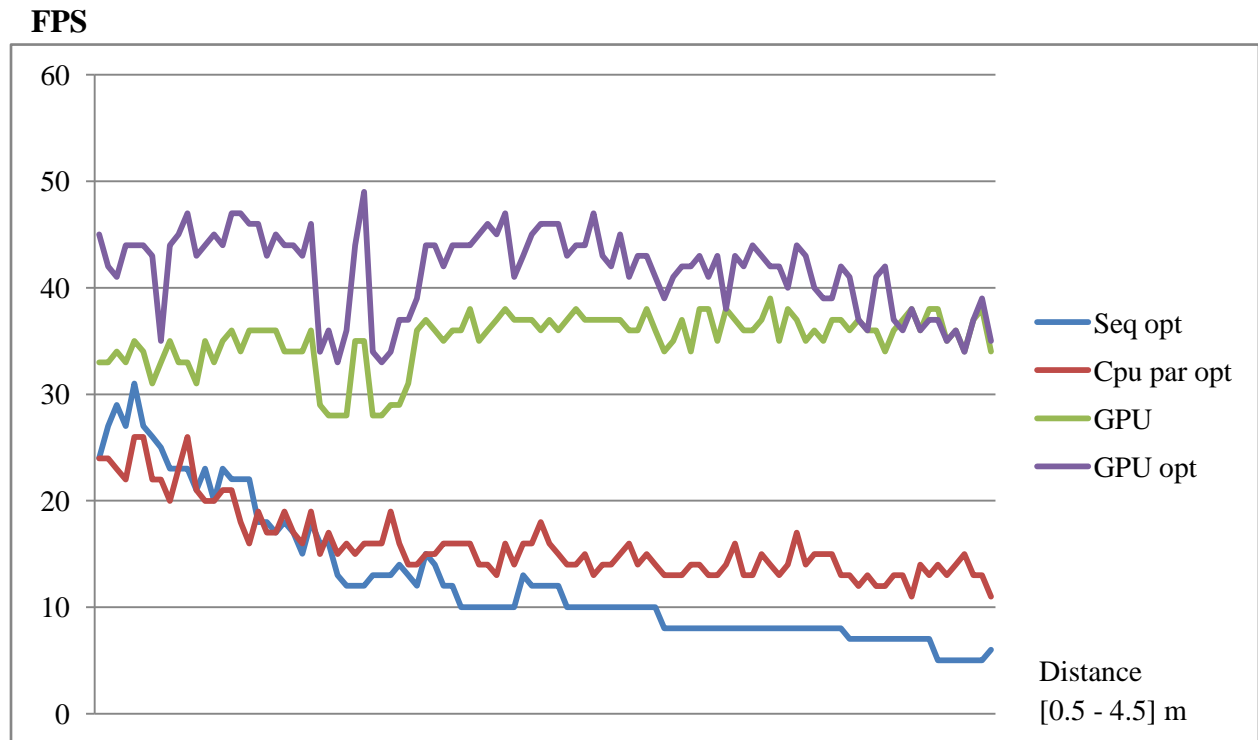


Figure 63: FACE STARTED TO GET FAR AWAY FROM THE CAMERA

The reason why the FPS decreases as we get far from the camera for the optimized versions is simple. The closer we are to the camera, the bigger is face and so the bigger is the detection window which implies less scales and less memory access.

The contrary is also true, the further we are from the camera, the smaller is the face and the detection window which implies a lot more scales to process and a lot more memory access.

The face sizes of the Fig. 63 can be seen in the Tab. 5 below in which they are ordered horizontally. The width and height sizes are the same.

Face Size							
222	221	228	212	210	197	192	191
180	176	163	149	142	143	141	145
132	137	133	125	123	120	117	118
116	113	105	107	111	110	108	106
104	95	93	98	97	96	86	87
88	89	90	95	85	84	83	85
79	77	76	75	74	73	70	72
67	68	69	66	64	62	63	61
60	59	57	58	56	55	54	53
52	51	50	49	48	47		

Table 5: Face Size

The Fig. 64 shows how the face size decreased with distance. We can see that the characteristics of the graph is not much different of the sequential and parallel CPU optimized versions.

### Face Size

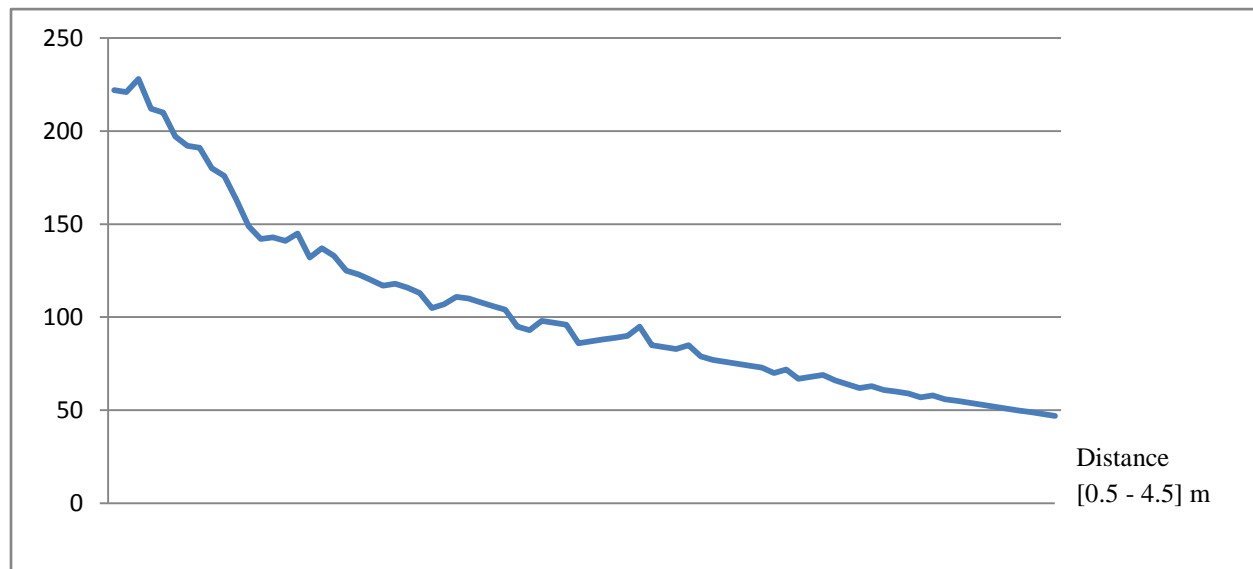


Figure 64: FACE SIZE DECREASED WITH DISTANCE

#### 7.4 Number of Sub-Windows Processed by the Stages of Cascade Classifiers

In order to perform this experiment we made use of IMG 3 which has a size of 2448x2448 pixels. The tests were made with the optimized versions. As mentioned on previous sections, the face cascade of classifiers has 22 stages.

When the algorithm is performed for the first time, before having any information of the size of the face, 46 scales are generated and a total number of 8 443 722 sub-windows have to be processed by the cascade of classifiers stages.

The second time the algorithm is performed, we already have the information about the face size, so taking advantage of the optimizations introduced for single face detection (Section 5.1.1), the number of scales was reduced to 7 and the number of sub-windows to 704.

Before optimization
  After optimization

Stage	Number of Sub-Windows Processed		Percentage % of Sub-Windows Processed		Number of Sub-Windows Discarded After Stage Processing	
0	8 443 722	704	100	100	1 871 798	72
1	6 571 924	632	77.84	89.78	2 955 953	213
2	3 615 971	419	42.83	59.52	1 606 683	141
3	2 009 288	278	23.8	39.49	1 101 759	95
4	907 529	183	10.75	26	390 541	23
5	516 988	160	6.13	22.73	237 060	26
6	279 928	134	3.32	19.04	95 409	16
7	184 519	118	2.19	16.77	58 081	14
8	126 438	104	1.5	14.78	62 849	10
9	63 589	94	0.76	13.36	27 521	6
10	36 068	88	0.43	12.5	15 699	1
11	20 369	87	0.25	12.36	10 317	8
12	10 052	79	0.12	11.23	4 206	3
13	5 846	76	0.07	10.8	2 078	10
14	3 768	66	0.05	9.38	1 681	7
15	2 087	59	0.03	8.39	844	1
16	1 243	58	0.02	8.24	470	1
17	773	57	0.01	8.1	291	2
18	482	55	0.01	7.82	159	5
19	323	50	0.01	7.11	102	0
20	221	50	0.01	7.11	67	1
21	154	49	0.01	6.97	0	0

Table 6: Sub-Windows Processed

## 7.5 Reliability Tests

The reliability tests were performed with a web camera having 0.3 megapixels which was able to provide frame sizes of 640 x 480.

### Moving Backwards

The goal of this test was to measure the maximum distance which is covered by our application. As is possible to realize from the Fig. 65, the facial features tend to not be detected as we move backwards.



Figure 65 a): MOVING BACKWARDS

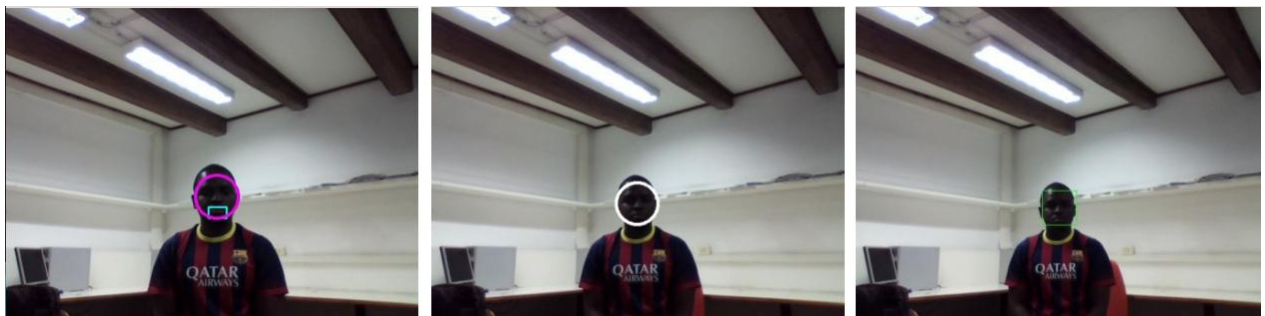


Figure 65 b): MOVING BACKWARDS

After a certain distance (~4.5 meters) the Viola-Jones failed to detect a face, the tracking algorithm (template matching defined in Section 4.2.1) is used as a backup when the detection procedure fails as it is possible to see in Fig. 65.

### Changing light conditions

For this case we started with the maximum light available in that room (Fig. 66 in the left) then decreasing the light (Fig.66 in the middle) and finally with the least light available (Fig. 66 in the right).

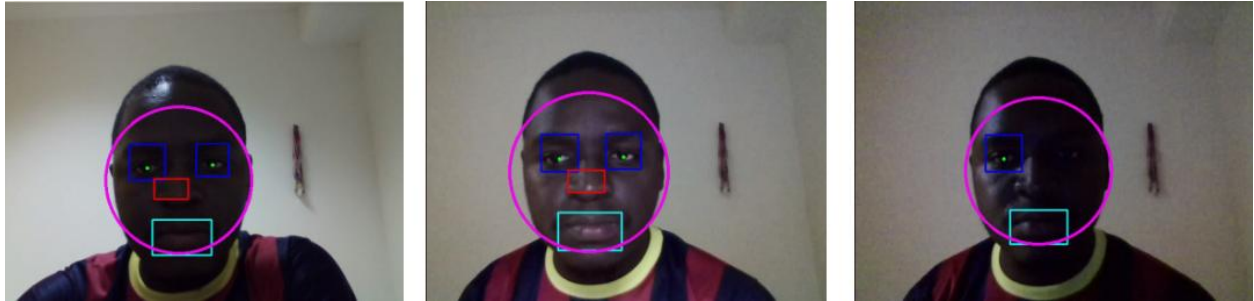


Figure 66: CHANGING LIGHT CONDITIONS

Is possible to notice, the detector is reliable for finding faces at different light environment while the facial features are the ones more sensible to the change of light.

### Surpassing the failure modes of Viola-Jones algorithm

For this test, we were focused on overcoming the failures mentioned on Section 3.5 with the solution proposed on 4.2. As it possible to notice from the Fig. 67, our methodology is able to find the faces even when they are rotated to a certain degree and also when they are occluded.

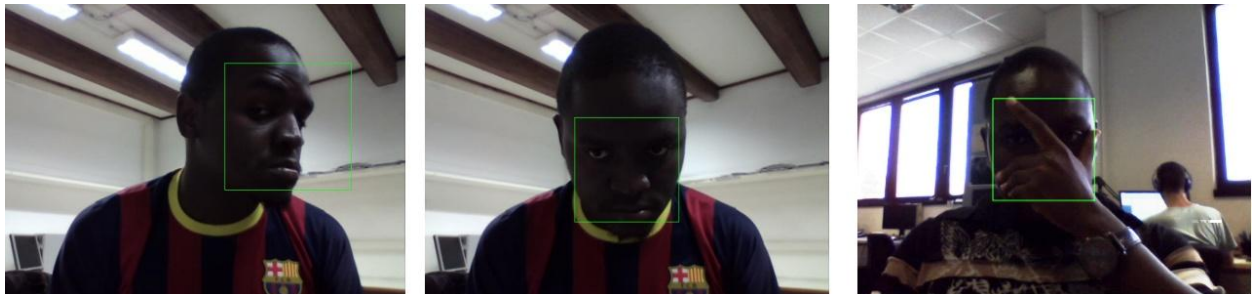


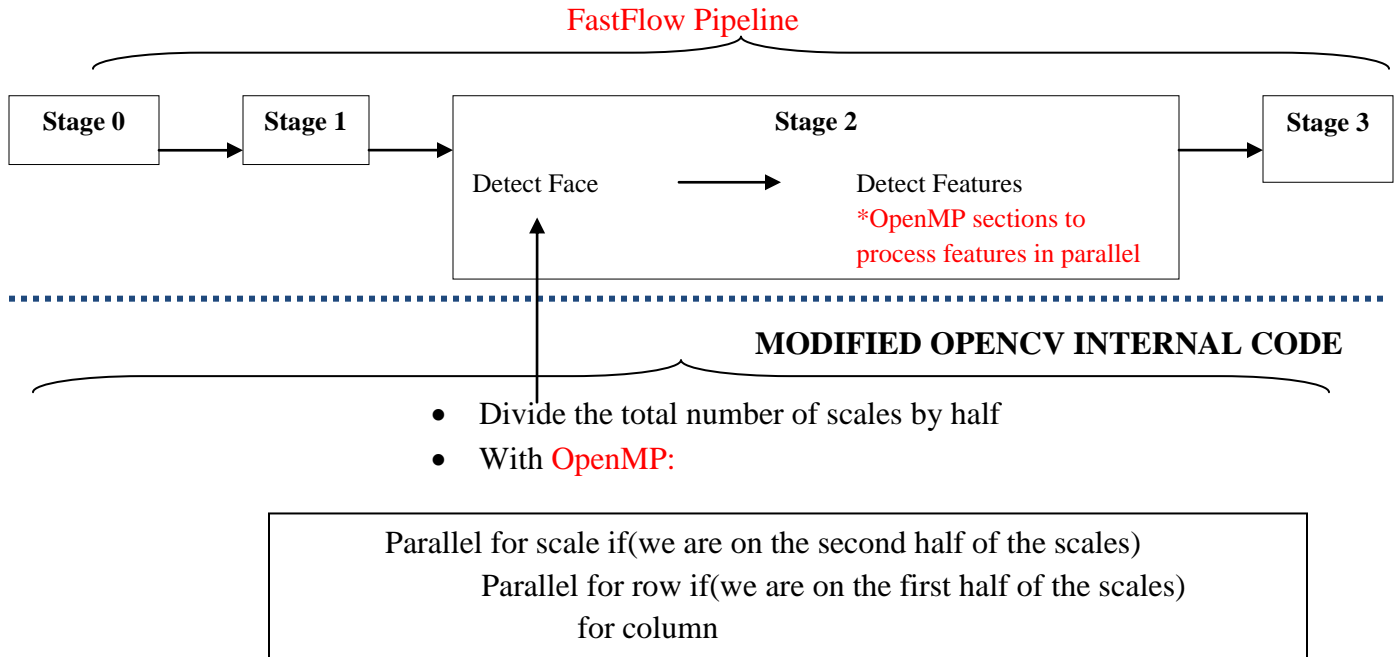
Figure 67: SURPASSING THE FAILURE MODES OF VIOLA-JONES ALGORITHM

## 7.6 Coexisting Different Parallel Programming Frameworks

Our parallel implementation on CPU Multi-Core Platform is based on two programming frameworks: FastFlow and OpenMP (Section 6.1).

FastFlow is utilized in order to implement the pipeline paradigm defined in Section 6.4 while OpenMP performs the heuristic solution (Section 6.4.1) to the OpenCV internal code and allows the features (eyes, mouth, nose) to be processed in parallel.

### APPLICATION CODE



If we do not specify the `-DNO_DEFAULT_MAPPING` option when compiling FastFlow code (as in our case), the FastFlow nodes are allocated from context 0 on [34]. This means that our four stages pipeline will be allocated such as CPU1 to stage 0, CPU 2 to stage 1, CPU 3 to stage 2 and CPU 4 to stage 3.

Taking advantage of the `GOMP_CPU_AFFINITY` option [33], we specified OpenMP to work on the remaining available resources (from CPU 5 to CPU 24, see Fig. 68).

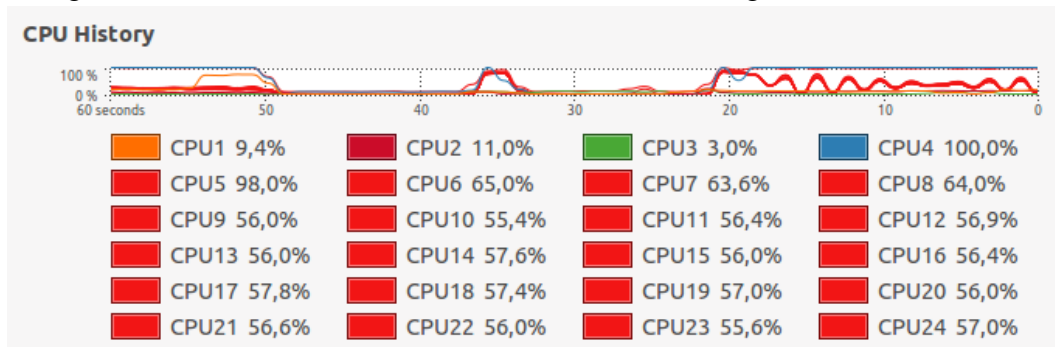


Figure 68: Exploiting all the available resources

## 7.7 Summary of the Results Achieved

In this section we have seen the tests performed to our application. It was demonstrated that there is no performance gain in separating face and features detection in a 5 stage pipeline or putting them together in a 4 stage pipeline. Based on those results, we decided to apply the 4 stage pipeline.

When testing the application with the Multiple Face Detection Approach specified on Section 6.4.1, the least size image (IMG 1) was able to achieve real-time processing with GPU (~38 FPS) and a quasi-real-time with the CPU parallelized version (~ 10 FPS) while the intermediate size image (IMG 2) was able to achieve a quasi-real-time processing with GPU (~ 17 FPS).

Applying the same experiments with the Single Face Detection Approach specified on Section 6.4.2, the least and intermediate images were able to achieve real-time processing with all the implementations apart from the originally sequential one. The largest image (IMG 3) was able to achieve real-time processing with the parallel CPU and GPU implementations.

When the application was tested on video frames, it was shown that the sequential and parallel CPU implementations optimized for single face detection are directly affected by the size of the face.

In section 7.4 was demonstrated that for the largest image size, the optimization introduced in Section 6.4.2 allowed to decrease the number of scales from 46 to 7 and the number of processed sub-windows from 8 443 722 to 704 which implies a reduction of 99.99% of processed sub-windows.

The reliability tests showed that our application is very reliable in different scenarios such as distance, light conditions, occlusion and rotation.

In Section 7.6 was demonstrated that it is possible to coexist the two parallel programming frameworks utilized in our application (FastFlow and OpenMP) in order to exploit all the available resources on CPU Multi-Core Platforms.

## Modifications to OpenCV Internal Code

In this Section, we will list all the functions and files that were modified or observed in order to implement new functionalities or have a better understanding of the OpenCV internal behaviour.

Function/File	Location	Lines added	Details
CVRunHaarClassifierCascadeSum	Opencv-3.1.0/modules/objectDetect/src/haar.cpp	5	Get the total number of processed sub-windows by each stage of the cascade of classifiers
CVHaarDetectObjectForROC	Opencv-3.1.0/modules/objectDetect/src/haar.cpp	33	<ul style="list-style-type: none"><li>- Get total number of scales to process</li><li>- Apply Heuristic solution of Section 6.4.1</li><li>- Get the processing time of each scale</li></ul>
CV::Parallel_For_	Opencv-3.1.0/modules/core/src/parallel.cpp	7	Know which parallel paradigm is implemented
CMakeLists.txt	Opencv-3.1.0/modules/core	1	Specify OpenMP as the parallel paradigm to implement
haarcascade_frontalface_alt.xml	Opencv-3.1.0/data/haarcascades	0	<ul style="list-style-type: none"><li>- Know how many stages face cascade has</li><li>- Know how many filters each stage has</li></ul>



## 8. Conclusions

The aim of this thesis was to Parallelize Image Processing Algorithms For Face Recognition on Multicore Platforms. In order to reach our goal we started by analyzing the challenges associated with face detection which involves factors such as pose, presence or absence of structural components, facial expression, occlusion and imaging conditions.

When the face detection methodologies were analyzed, was concluded that the ones based on learning algorithms (appearance based) allows to achieve better results due to the fact that eliminates the potential modeling error as a result of an incomplete or inaccurate face knowledge.

Among the state-of-the-art algorithms for face detection, Viola-Jones has an edge as it is very solid since the original paper still the main source of reference. This algorithm is able to process images extremely rapidly based on three innovative contributions: The integral image, the AdaBoost learning algorithm and a cascade of classifiers.

The Viola-Jones algorithm is still not optimal as the detector becomes unreliable for faces with a certain rotation, the detector may fail when the faces are very dark while the background is relatively light and the detector fails on significantly occluded faces.

In order to overcome those failures we proposed a solution based on a tracking algorithm which proved to be very efficient although it is still not optimal as it first needs the Viola-Jones algorithm to know what to track.

When implementing the Viola-Jones algorithm with OpenCV, it was shown that there are two histogram equalization methods that are optimal in different scenarios. An algorithm was proposed in order to apply the best method on each specific scenario.

In order to precisely detect a face, was shown that we need to verify how many facial features are present in the supposed face found. When determine if the eyes are open or not the most effective solution is to apply two cascades, first the one that recognizes both open and closed eyes, second the one that recognizes only open eyes. The outcome of the second cascade will determine if eye is open or not.

In order to render the face detection process faster two optimizations were proposed. The first optimizes the face detection task based on the size of the face found on previous frame. This optimization allows decreasing drastically the computation time. The drawback of this optimization is that it should be applied only for one face, or at most multiple faces that are somehow equidistant from the camera.

The second optimization also allows decreasing the computation time as the size of the facial features to look for is based on a ratio with the size of the face found.

In order to parallelize the entire face detection process, it was shown that the pipeline paradigm is the most suitable solution as the whole process is performed on a sequence of phases.

When parallelizing the face detection task individually, with a limited number of resources, dividing the computation space into a collection of detection windows is the most valuable parallelization strategy. An optimization was proposed to the current OpenCV parallel implementation.

In order to take advantage of the face detection parallelization on GPUs, [22] specifies that we need to minimize accesses to device memory, the transfers to and from the GPU must be overlapped with computation, memory accesses to the GPU global and shared memory must avoid bank conflicts and branching should be reduced to a minimum within kernels.

Our tests have demonstrated that our application is able to achieve real-time processing rates at different scenarios and it is reliable in various situations.

In Section 6.6 was presented a theoretical study on the reasons why the Viola-Jones algorithm may not allow multi-core platforms to be exploited to their full capacity. For the GPU case we based our analysis with the studies made by [35] which points three main levels of unbalance such as Thread Level unbalance, Block Level unbalance and Image Level unbalance.

In order to solve those problems, [35] proposes five main optimization techniques such as: Warp Front Size Work Granularity, Persistent Threads, Uberkernel, Local Queues and Global Queues. These techniques should help algorithms whose memory access and work distribution are irregular by eliminating unbalanced computation in order to improve the performance on GPUs.

For the CPU case, we proposed a combined solution based on two parallelization strategies. The first applies a three stage pipeline based on the percentage of memory access and processing load of the stages of the cascade of classifiers.

The second, implements a dynamic technique such that when a thread process has finished computing its entire assigned task(s) (e.g a row of the image), instead of becoming idle, it can randomly choose one of the active working thread and steal some task to be computed from those previously assigned to that thread.

Future work includes study in more details the aspects presented on Section 6.6 and implement them in order to verify their efficiency. Also a more detailed analysis in some of the principles mentioned by [22] in order to take full advantage of the GPU hardware requires further investigation.

## 9. References

- [1] E. Hjelm and B. K. Low. "Face Detection: A Survey", Computer Vision and Image Understanding, vol. 83, 2001, pp. 236 – 274.
- [2] M. Yang, D. Kriegman, N. Ahuja. "Detecting faces in images: a survey", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24, No 1, 2002, pp. 35–58
- [3] Giorgio Buttazzo. "Hard Real-Time Computing Systems", Third Edition, Springer, 2011
- [4] Yang, Shuo, et al. "From facial parts responses to face detection: A deep learning approach", Proceedings of the IEEE International Conference on Computer Vision, 2015.
- [5] Henry A. Rowley, Shumeet Baluja and Takeo Kanade. "Neural Network-Based Face Detection", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, No 1, 1998, pp. 23-38
- [6] P. Viola and M. Jones. Robust Real-Time Face Detection, International Journal of Computer Vision, vol. 57, No 1, 2001, pp. 137-154.
- [7] Astha Jain et al. "Improvements in OpenCV's Viola Jones Algorithm in Face Detection - Tilted Face Detection", Int. J. of Signal and Image Processing, vol. 5, 2014.
- [8] Rainer Lienhart and Jochen Maydt. "An Extended Set of Haar-like Features for Rapid Object Detection". IEEE ICIP, vol. 1, 2002, pp. 900-903.
- [9] Harry Commin. "Robust Real-time Extraction of Fiducial Facial Feature Points using Haar-like Features", Master Thesis, Imperial College London , 2009
- [10] Kalinovskii I.A. and Spitsyn V.G. "Compact Convolutional Neural Network Cascade for Face Detection", CEUR Workshop Proceedings, 2015, pp. 375-387
- [11] OpenCV. "The OpenCV Reference Manual", Release 2.4.9.0, 2014.
- [12] Histogram Equalization of Grayscale or Color Image, accessed 04/16, <http://opencv-srf.blogspot.it/2013/08/histogram-equalization.html>
- [13] Histograms - 2: Histogram Equalization, accessed 04/16, [http://docs.opencv.org/3.1.0/d5/daf/tutorial\\_py\\_histogram\\_equalization.html#gsc.tab=0](http://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html#gsc.tab=0)
- [14] Ashwith Kumar, Suresh and Sanjeev Kubakaddi. "Multiple Face Detection and Tracking using Adaboost and Camshift Algorithm", Int. Journal of Research Studies in Science, Engineering and Technology, vol. 1, 2014, pp 8-15.

- [15] Gi-Woo Kim, Dae-Seong Kang. “Improved CAMshift Algorithm Based on Kalman Filter”, Advanced Science and Technology Letters, vol.98, 2015.
- [16] Marco Vannesch. “High Performance Computing: parallel processing models and architectures”, Pisa University Press, 2014
- [17] Marco Danelutto. “Distibuted Systems Paradigms and Models”, Teaching material of Laurea Magistrale in Computer Science and Networking, 2015.
- [18] NVidia. “What is GPU Computing”, accessed 05/16, <http://www.nvidia.com/object/what-is-gpu-computing.html>
- [19] Roberto Ladu. “GPU: A Formal Introduction”, LIST Group S.p.A. and Department of Computer Science University of Pisa, 2012.
- [20] Kayvon Fatahalian. “How a GPU Works”, Presentation, Carnegie Mellon University School of Computer Science, 2011.
- [21] Bharkat Kumar Sharma et al. “Towards a Robust, Real-time Face Processing System using CUDA-enabled GPUs”, Conference: High Performance Computing (HiPC), 2009.
- [22] Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh et al. “Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs”, Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium
- [23] Cornell University Center for Advanced Computing. “Memory Coalescing”, accessed 05/2016, <https://cvw.cac.cornell.edu/gpu/coalesced?AspxAutoDetectCookieSupport=1>
- [24] N. Prabhaka, V. Vaithyanathan, Akshaya Sharma, et al. “Object Tracking Using Frame Differencing and Template Matching”, Research Journal of Applied Sciences, Engineering and Technology, 2012.
- [25] Lam K, Yeong Y, Yew C et al. “A Study on Similarity Computations in Template Matching Technique for Identity Verification”, International Journal on Computer Science and Engineering, vol. 02, 2010
- [26] Kari Pulli, Anatoly Baksheev, Kirill Korniyakov, et al. “Realtime Computer Vision with OpenCV”, ACM Queue, 2012.
- [27] Joshua Miguel, Jordan Saleh, Michelle Wong. “Parallelizing Face Detection in Software”, Department of Electrical and Computer Engineering, University of Toronto, 2013.
- [28] Khalil Khattab, Philippe Brunet, Julien Dubois, et al. “Real Time Robust Embedded Face Detection Using High Level Description”, University of Burgundy, France, 2011

- [29] Massimo Torquati. "Very short Tutorial: The FastFlow programming model", accessed 09/2016, <http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:tutorial>
- [30] Tim Mattson, Barbara Chapman. "OpenMP\* in Action", accessed 09/2016 <http://openmp.org/wp/presos/omp-in-action-SC05.pdf>
- [31] FastFlow (FF), accessed 09/2016 <http://calvados.di.unipi.it/dokuwiki/doku.php/ffnamespace:about>.
- [32] Joel Yliluoma. "Guide into OpenMP: Easy multithreading programming for C++", 2007, accessed 09/2016, <http://bisqwit.iki.fi/story/howto/openmp/>
- [33] Glenn K. Lockwood. "Managing Process Affinity in Linux", accessed 09/2016, <http://www.glennklockwood.com/hpc-howtos/process-affinity.html>
- [34] Marco Danelutto, Massimo Torquati. "Structured parallel programming with core FastFlow", Central European Functional Programming School, 5th Summer School, pag. 29-75, 2015
- [35] Haipeng Jia, Yunquan zhang, Jianliang Xu, Weiyan Wang. "Accelerating Viola-Jones Face Detection Algorithm On GPUs", IEEE 14th International Conference on High Performance Computing and Communications, 2012