An Efficient GPGPU based Implementation of Face Detection Algorithm using Skin Color Pre-treatment

Imene Guerfi[a], Lobna Kriaa[b] and Leila Azouz Saidane
CRISTAL Laboratory, RAMSIS Pole, National School for Computer Sciences (ENSI), University of Manouba, Tunisia

Keywords: GPU Computing, Parallel Programming, Face Detection, AdaBoost, Haar Cascade, Skin Color Segmentation.

Abstract: Modern and future security and daily life applications incorporate several face detection systems. Those systems have an exigent time constraint that requires high-performance computing. This can be achieved using General-Purpose Computing on Graphics Processing Units (GPGPU), however, some existing solutions to satisfy the time requirements may degrade the quality of detection. In this paper, we aimed to reduce the detection time and increase the detection rate using the GPGPU. We developed a robust, optimized algorithm based on an efficient parallelization of the face detection algorithm, combined with the reduction of the research area using a mixture of two color spaces. for skin detection. Central Processing Unit (CPU) serial and parallel versions of the algorithm are developed for comparison's sake. A database is made using a classification method to evaluate our approach in order to discuss all scenarios. The experimental results show that our proposed method achieved 27,1x acceleration compared to the CPU implementation with a detection rate of 97,05%.

1 INTRODUCTION

In the last decade, the identification requirements of each individual are increasing rapidly, especially in security and daily life applications. Security experts have suggested biometrics as an efficient method to ensure security. The trend started with fingerprints and now it is shifting towards facial recognition (Khan et al., 2017). The report published by Market research Future predicts that the global facial recognition market will increase considerably during the forecast period (MRF, 2019) due to the growing need for surveillance and security as a result of increased criminal activities. Mobile facial identification and the rise in popularity of smart homes are also fueling this growth of the facial identification market.

A face processing system comprises face detection, recognition, tracking, and rendering. The primary and substantial aspects of any face processing system are face detection. It is used to detect the presence and the precise location of one or more faces in a digital image or video sequence. Recently face detection has received significant attention in academia and industry, mainly due to its wide range of applications, such as public security, video conferencing, entertainment and human-machine interface. Several of these applications are interactive and require reliable and fast face processing. Paul Viola and Michael Jones realized the first real-time face detection algorithm (Viola and Jones, 2001). Until today, the viola-jones algorithm has been widely applied in digital cameras and photo organization software.

The research on face detection, for the most part, has been focused on designing new algorithms or improving the detection rate and decreasing the false positive rate of the existing methods. Therefore, the majority of the available works are software solutions designed for general-purpose computational processors (GPP) (Bilaniuk et al., 2014). However, detecting faces in images is a computationally expensive task; hence, we need to look for high-performance solutions for fast face detection at reasonable cost. Recently, processor performance has evolved by increasing the number of computing units. In particular, the graphics-processing unit (GPU) was used in collaboration with the CPU to accelerate high computational general-purpose applications, by offloading compute-intensive portions of the application to the GPU. At the same time, the rest of the code still runs on the CPU. Heterogeneous computing that combines traditional processors coupled with GPU has become a promising solution in most systems to achieve higher performance. The GPUs, in general, are used to accel-

[a] https://orcid.org/0000-0002-2886-1713
[b] https://orcid.org/0000-0002-2112-7807

Guerfi, I., Kriaa, L. and Saidane, L.
An Efficient GPGPU based Implementation of Face Detection Algorithm using Skin Color Pre-treatment.
DOI: 10.5220/0009833305740585
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 574-585
ISBN: 978-989-758-443-5

erate applications as they integrate hundreds of cores designed to handle many simultaneous tasks.

Various works studied how to redesign the face detection algorithm for GPU parallel execution optimally. Many promising results have been presented and a lot of works claim to offer the fastest GPU implementation. However, the efficiency of face detection algorithms does not depend only on the speed of detection, because this last can be affected easily by the change of some parameters such as the number of the classifiers, the step of scanning, the scaling factor, the size of images and the number of faces in the image (Wei and Ming, 2011). To provide an efficient face detection algorithm the detection rate and speed should not be decreased and false-positive should not be increased.

In the present work, we present an efficient GPGPU based implementation of the Viola-Jones face detection algorithm using the "Compute Unified Device Architecture" (CUDA) programming model. For the time acceleration and the improvement of the detection rate, we propose a new optimized parallel method combined with a research area reduction using a skin color pre-treatment. In addition, an upscaling for the size of the small images was done to improve the detection rate. For comparison reasons, a CPU single-threaded and CPU multithreaded version of the code was developed. Standard and stable parameters and a lot of image sizes with variant face numbers were used for all the implementations so that the evaluation could be more effective.

We evaluate our efficient face detection algorithm using a created database composed of 700 color images taken from the web. The experimental results indicate that our parallel face detector achieves 27.1x and 18x speedup compared to CPU single-threaded and multi-threaded versions respectively while increasing the accuracy and reducing false-positive rate.

The rest of this paper is organized as follows: The next section gives an overview of the related work. Section 3 describes the Viola and Jones algorithm. Section 4 gives details of the proposed optimized algorithm for face detection. Section 5 elaborates experimental results and discussion, and finally. Section 6 presents the conclusion.

2 RELATED WORKS

Face detection was one of the first computer vision applications; the search in this field was begun since the middle of the 1960s. Indeed, the majority of the early works did not propose efficient methods (Zafeiriou et al., 2015). In 2001, a revolution in this field was made when Viola and Jones invented a reliable face detection technique with promising accuracy and high-efficiency(Viola and Jones, 2001). The used algorithm was based on Adaboost training and Haar cascading. It achieved an average of about 15 frames per second (fps) for a (320x288) image. It made face detection practically feasible in real-world applications. Until today, this algorithm has been widely applied in digital cameras and photo organization software.

Since then, many relevant works have been presented for accelerating the Viola-Jones face detection algorithm. The available computational resources limited earlier implementation of this algorithm. However, with the increase of low-cost, high-performance computational devices, many researchers have begun to explore the usage of these features.

(Sharma et al., 2009) introduced the first GPU realization of a face detection algorithm using CUDA. They reached a detection at 19 fps on a (1280 × 960) video stream, which is a good improvement in detection time. However, the accuracy was only 81% with 16 false positives on the CMU test set. (Kong and Deng, 2010) proposed a GPU accelerated OpenCV implementation that achieved between 49.08 ms and 196.73ms (20,4-5,1 fps) on images from (340x240) to (1280x1024). (Hefenbrock et al., 2010) presented a multi-GPU implementation. They used a desktop server containing four Tesla GPUs for the implementation and achieved 15.2 fps. However, the integral image computation was not parallelized. In their other works, (Nguyen et al., 2013) they used 5 Fermi GPUs and improved in efficiency by using a dynamic warp scheduling approach to eliminate thread divergence. They used the technique of thread pool mechanism to significantly alleviate the cost of creating, switching, and terminating threads. They reported realized 95.6 fps on (640x480) images. The proposed approach (Devrari and Kumar, 2011) includes enhanced Haar-like features and uses SVM (Support Vector Machine) for training and classification. They achieved 3.3 fps on (2592x1900) images. The (Wei and Ming, 2011) implementation reached 12 fps on (640*480) images. However, the used method causes inadequate usage of resources. A Haar-based face detection for (1920x1080) video on GTX470 was proposed by (Oro et al., 2011) (Oro et al., 2012) and achieved a performance of 35 fps. (Tek and Gökmen, 2012) used 3 GPUs for the implementation and they achieved 99 fps with good detection rate even though the classifier was small.

Other relevant implementations of the algorithm are found in the work of (Jeong et al., 2012), (Li et al., 2012)and (Sun et al., 2013), whose experiments reveal improve detection speed for high-resolution images. Other works (Meng et al., 2014) (Bhutekar and Manjaramkar, 2014) focused on improving detection for small images.

(Chouchene et al., 2015) proposed parallel implementation of face detection on (NVidia310M) GPU that could achieve 24 fps for small images (32x32) and only 11 fps for bigger images (1024x1024). (Wai et al., 2015) presented Open CV accelerated implementation using the latest GPU at that time (Tesla k40) and they achieved 37.91 fps on (640x480) images. (Fayez et al., 2016) proposed an image scanning framework using GPGPU in which they have implemented the Viola-Jones algorithm. They have achieved 37fps for (1920x1080) images. In (Jain and Patel, 2016) introduced an implementation that increases the speed of the algorithm 19.75x over a CPU implementation. (Mutneja and Singh, 2018) presented face detection using a combination of motion and skin color segmentation, the test samples were low-resolution videos (600x800) and they get 3.16 fps. In their other work (Mutneja and Singh, 2019) much more analysis was done for the algorithm and they achieve 25 fps for (480x640) images. (Patidar et al., 2020) presented an optimized parallel face detection system using CUDA on GPU and achieved 1.28 fps in the FDDB image set. Although interesting results were recorded, a lack of information about the used classifier and accuracy and false-positive and the test data set in most of the works.

3 BACKGROUND

3.1 Viola-Jones Algorithm

The Viola-Jones Face detection algorithm is one of the best face detection algorithms that have been developed through time. The algorithm is an appearance-based model, and it can be divided into two phases. A training phase where a cascade classifier is generated based on a set of positive and negative samples. It is a concatenation of several weak classifiers divided into stages that get increasingly complex. With the AdaBoost algorithm for the selection of informative human facial features. The second is the detection phase, where the algorithm will detect faces in a given image using the pre-trained cascade classifier. A small window will scan the image, at each position the cascade classifier is applied. To reduce the treatment if a window doesn't pass a stage of the cascade classifier is directly rejected. Otherwise, it passes to the next stage; if it passes all stages, then it contains a face. This algorithm consists of four parts:

3.1.1 Haar-Like Features

Human faces share some similar properties. These properties are mapped mathematically to the Haar features (Figure 1). A Haar feature of a rectangle $h(r)$ is a scalar calculated by summing up the pixels in the white region $p(w)$ and subtracting those in the dark region $p(b)$ (equation 1).

$$h(r) = \sum p(w) - \sum p(b) \qquad (1)$$

These features are used in the training step to form the classifiers and in the detection step. When an image is scanned to detect a face, at each step, the features in the actual window are compared to the trained one.

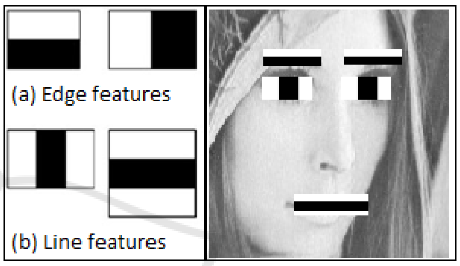


Figure 1: Fore types of Haar features with example.

3.1.2 Integral Image

The integral image was proposed to overcome the huge amount of calculation caused by identifying Haar features. The integral value I at pixel (x,y) is the sum of all the pixels above it and to its left (equation 2).

$$I(x,y) = \sum_{\acute{x} \leq x, \acute{y} \leq y} i(\acute{x},\acute{y}) \qquad (2)$$

This allows very fast feature evaluation, since calculating the sum of pixels inside a rectangle requires the Integral image values of the four corners only (equation 3), which leads to computing features in constant time efficiently.

$$h(x\acute{x}y\acute{y}) = I(\acute{x},\acute{y}) - I(x,\acute{y}) - I(\acute{x},y) + I(x,y) \qquad (3)$$

3.1.3 AdaBoost

Adaptive Boosting is a machine learning algorithm. They are applied in the training phase to select the features that best describe a face. This algorithm combines these features into weak classifiers, and then it groups them to create strong classifiers that form the cascade classifier.

3.1.4 Cascade Classifier

The cascade classifier is generated during the training phase and used in the detection phase. During the detection, a sub-window scanned the image and tried at each step to determine if the current window could be a face. To ensure that each window should pass through the pre-trained cascade classifier. Efficient cascade classifier constructed to reject the maximum of the negative windows at early stages to reduce the computation time (figure 2).

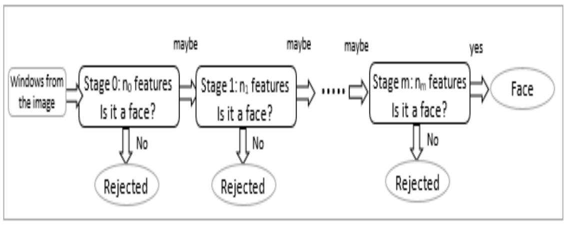

Windows from the image
Stage 0: n_0 features Is it a face?
maybe
Stage 1: n_1 features Is it a face?
maybe
maybe
Stage m: n_m features Is it a face?
yes
Face
No
No
No
Rejected
Rejected
Rejected

Figure 2: A cascade classifier.

Because the focus of this paper is the parallel acceleration of the face detection process, we obtain the classifier not by self-training, but by using the OpenCV open-source software (OpenCV,). The classifier has 2913 features divided into 25 stages (Table 1), with min windows size of 25x25.

Table 1: The used cascade classifier.

stages	0	1	2	3	4	5	6	7	8	9	10	11	12
features	9	16	27	32	52	53	62	72	83	91	99	115	127
stages	13	14	15	16	17	18	19	20	21	22	23	24	total
features	135	136	137	159	155	169	196	197	181	199	211	200	2913

4 PROPOSED ALGORITHM

This section describes our optimized face detection algorithms based on an efficient parallel implementation using CUDA and integrating of RGB and YCbCr color spaces to reduce the research area. Before that, the single-threaded version is presented to make the algorithm clear and for comparison reasons.

4.1 Sequential Algorithm

Face detection implementation comprises 4 main steps: 1) resizing of the original image into a pyramid of images at different scales 2) calculating the integral images for fast feature evaluation, 3) Computing the image coordinates for each Haar feature, and 4) detecting faces using a cascade of classifiers. Figure 3 shows the overall flow of the data. The process starts with reading the input image and loading the cascade classifier. After that, the image is converted to grayscale and the new height and width are calculated to resize the image. The algorithm consists of resizing the image with scale factor 1 for the first time than with a predefined scale factor, until the image is equal or smaller than the detection windows 25x25 (the size obtained from the used classifier), in our case the scale factor is 1,2. Each time the image is resized, we transform it into an integral image format and we also calculate the integral image of the square root of the intensities of the pixels to accelerate the further calculation. The next step is to compute the image coordinates for each Haar feature, these are the relative positions of each Haar rectangle boundary in a 25x25 window. For a shifted position of the detection window, the shifted offset is added to get the new coordinates. After that, the window passes through the stages of the cascade classifier. At each stage, if the integral sum is less than the threshold, this window is rejected. Otherwise, the window passes to the next stage. If it completes all the stages then it is stored as a face. Then the window is shifted by a step of one pixel and passed through the cascade classifier. When the window scans all the image, the image is resized again and processed through the same steps until the condition is satisfied. Finally, the stored faces are indicated by a circle around them in the original image.

4.2 GPGPU Implementation

The proposed research work is the detection of multiple human faces from images with different sizes using Haar-features and cascade classifier, employing skin color pre-treatment for search space reduction and GPU acceleration for faster processing.

In this section, we introduce some of the main ideas for parallelizing face detection exploiting both CPU and GPU. First of all, when the image is loaded it is directly transformed into grayscale and save to the GPU global memory. The GPU doesn't transfer the result to avoid the communication overhead. The cascade classifier is also saved to the GPU global memory as multiple vectors, for example, one for the number of features in each stage, one for all the rectangles coordinate and anther one for the threshold. This part of the algorithm is handled by the CPU. The parts that can be parallelized in this algorithm are image resizing, integral image computation, the compute of image coordinates for each Haar feature and scanning the image and processing each window by the cascade classifier. In what follows, we explain each one separately.

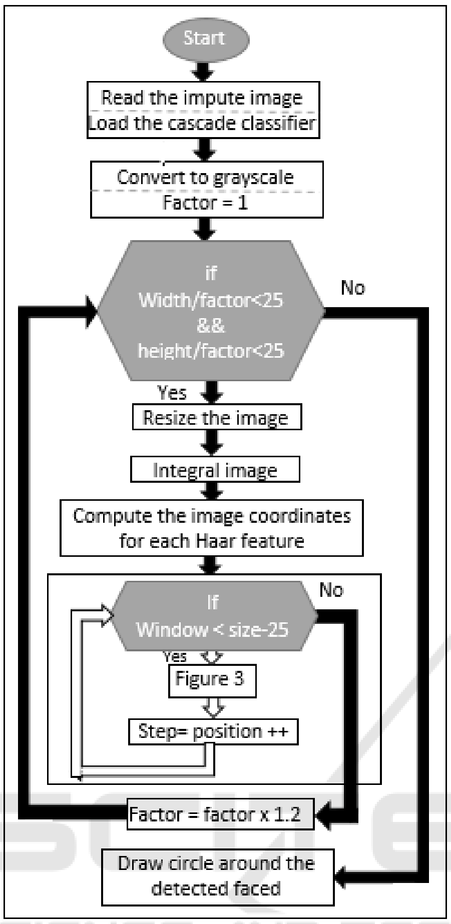


Figure 3: The face detection algorithm.

4.2.1 Image Resizing

Since the size of faces in the image could be variante and the detection window is stable (25x25), the image should be resized until it is equal to the detection window size. for that, we use the nearest neighbor algorithm to create a pyramid of images at different scales (figure 4). The nearest neighbor is the simplest and fastest implementation of the image scaling technique (Jiang and Wang, 2015). It is very useful when speed is the primary concern.

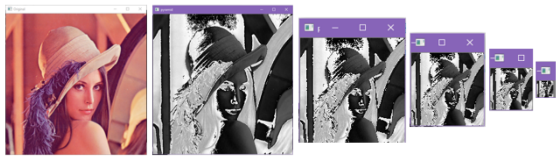


Figure 4: Image pyramid.

In this step the image width and height are downscaled by a factor of 1.2, in order to ensure that, we need to create a new image and compute both the horizontal and vertical ratios between the original image and the new image, after that, we delete redundant pixels based on these ratios. After analyzing the algorithm we can see that computing is independent, so we can map each pixel position to be fetched by a single thread. The total number of threads should be equal to the new image (width x height).

4.2.2 Integral Image

After image resizing, we have to compute the integral image, which is computationally expensive, especially for large images, because the value of each pixel is the sum of all the pixels above and to the left of it. To parallelize this step we have to remove the dependency between data. The main idea is to split the algorithm into 2 parts (Figure 5), the first is the sum of each row independently of others and the second is the sum of the columns. For that, we need as much thread as the number of rows, and we have to create a vector to hold the intermediate image. Here, each thread processes each row separately, it adds the previous pixel value, to the current pixel value along that row.The vertical sum computation is done on the output of the horizontal sum computation. It is similar to the horizontal sum; however, the threads compute the sum over the columns. The results are saved in a new image.

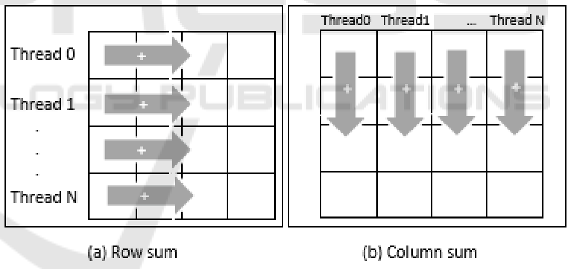


Figure 5: Image integral computation.

Similarly, we compute the integral sum of the squared pixel values, that we need to calculate the variance of the pixels for the Haar rectangle coordinates, in the Cascade classifier stage. We calculate the integral image and the integral image of the squared pixel at the same time since they use the same data.

4.2.3 Preparing the Image Coordinates

Next, we compute the image coordinates for each Haar feature, and these are the relative positions of each HAAR rectangle boundary in a (25x25) window. This calculation is a preparation for the next step when we shift the detection window. We will just need to shift the offset to get the new coordinates. The cascade classifier contains a 2913 feature each of which has 3 rectangles. We processed each feature separately, which means that each thread will process 3 rectangles.

4.2.4 Scanning the Image

In order to carry the detection, in the last step, we have to place a detection window (25x25) over the output of the previous step. We process through the cascade classifier, then we slide the window to the next position, until the end of the image. The cascade classifier processing can't be split because at each stage, if the window is rejected there is no need to process the rest, so this part should be done together. However, each window computation is independent of the others. For that, each scan window is processed independently. Hence, one thread is assigned to process a scan window.

This is the simple parallel implementation of the algorithm. In the following, we will present the optimization added to accelerate the process.

4.3 Optimization

The objective of parallel computing is to reduce the calculation time of a process. The GPU architectures are increasingly used for that purpose. To exploit the GPU performance it is essential to know the properties of the hardware architecture. The efficiency of an algorithm implemented on a GPU is related to how the available resources are used. In this subsection, we will introduce the optimizations and how to exploit the resources better. In what follows, the optimization in each part of the algorithm is explained separately.

4.3.1 Image Resizing and Sum of Rows

We merge the compute of image resizing and the first part of the integral image (row sum) in the same kernel, to avoid storing the resized image to global memory between kernel. In this case, each thread processes one row separately, when it computes a pixel value with the nearest neighbor it adds it to the previous pixel values and it saves it to an intermediate vector. To facilitate the next calculations, we save the values in the column that have the same number as the computed row, so that in the next step when we calculate the sum of the column, this latter will be the row with the same index as the column. We transpose it again after the column sum. This optimization helps us to avoid global memory access, that could cost 200-800 clock cycles on an NVIDIA. Therefore, memory optimization is essential in GPU based parallel face detection and it improves processing performance.

4.3.2 Sum of Columns

After the end of the first kernel, the result stays in the GPU global memory. We invoke 2 kernels, the first for the computation of the sum of columns and the second for the computation of the sum of the columns for the square value of pixels. Since we don't need the second kernel until the image scanning, we execute it asynchronously on a different stream, and we left the first kernel to be treated by the default stream. We left the second kernel to be executed concurrently with the kernel that prepares the image coordinates for each Haar feature. Finally, we synchronize all the kernels before the image scanning.

4.3.3 Scanning the Image

As we mentioned before (section 4.3.4). In this step, we process each window with one thread through all the features of the cascade classifier. These features are common to all the scan windows; hence we import them to the shared memory so that they can be visible to all the threads of the same block and like this, we can avoid multiple global memory access. Shared memory has lower latency and higher bandwidth than global memory, for this we use it to improve the performance. The features need nearly 69 KB of memory, however, in our case, the used GPU has only 48 KB of shared memory. For this purpose, we decide to import only the most used features so that we reduce as much memory transactions as possible. We mention here that the newer NVIDIA architecture has more shared memory, and make the import of all features in shared memory possible.

4.3.4 Overlap Data Transfers

The last optimization is to overlap data transfers when we copy the detected face vector to CPU memory. For that, we divided the data into multiple chunks. We execute each chunk on different stream and we copy the data of each streams separately.

The latter approach was further improved by applying skin color filtering to reduce the search space. This skin color filtering will be discussed in the next section.

4.4 Skin Color Segmentation

Skin color segmentation can be accomplished by explicitly modeling the skin distribution of certain color spaces using parametric decision rules (bin Abdul Rahman et al., 2007). A literature survey shows that different color spaces are applied for skin color analysis. In most cases, the default color space is the well-known RGB, which is composed of three primary colors, red, green and blue. The variations in skin color pixels due to illumination levels are minimized when utilizing the RGB color-space. However, due to intermixed chrominance (color information) and luminance (brightness measurement), it is least recommended for color tone analysis (Shifa et al., 2020). Another well used color space is YCbCr. In this space, the intensity of light is represented by luminance (Y) and chrominance is found by calculating the blue (Cb) and red (Cr) differences relative to luminance. The experimental result of (Shaik et al., 2015) shows that YCbCr color space can be applied for complex color images with uneven illumination.

In the case of real scenario, illumination variation remains a challenging task. For this, we utilize the additional luminance and chrominance information of the image on top of standard RGB properties to improve the skin pixels segmentation. We used the RGB boundary rules introduced by (bin Abdul Rahman et al., 2007) (equation 4) and the suitable ranges of YCbCr introduced by (Saikia et al., 2012) (equation 5). Equation 6 presents the combination to detect skin color, the other pixels are colored by the black color. The results are shown in figure 6, where (a) is the original image, (b) is the skin color segmented image and (c) is the gray scale of (b).

The skin colour at uniform daylight illumination rule is defined in RGB as:

$$\begin{aligned}&(R > 95)\ AND\ (G > 40)\ AND\ (B > 20)\ AND\\&(max\{R,G,B\} - min\{R,G,B\}\ > 15)\quad AND\\&(|R-G| > 15)\ AND\ (R > G)\ AND\ (R > B)\end{aligned}\tag{4a}$$

while the skin colour under flashlight or daylight lateral illumination rule in RGB is given by:

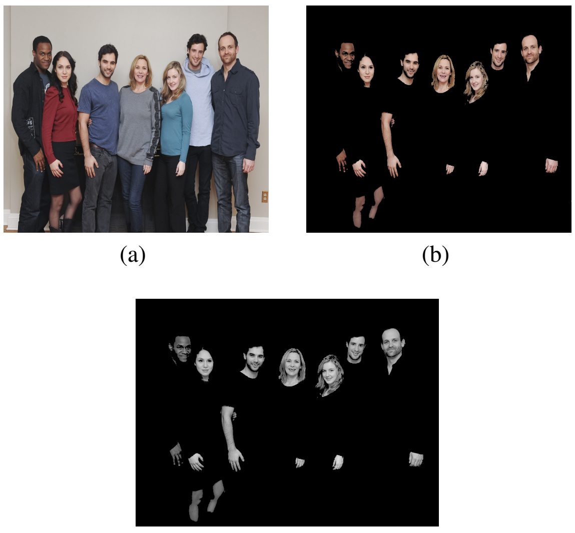
$$\begin{aligned}&(R > 220)\ AND\ (G > 210)\ AND\ (B > 170)\ \ AND\\&(|R-G| \leq 15)\ AND\ (R > B)\ AND\ (G > B)\end{aligned}\tag{4b}$$

$$(4a)\ OR\ (4b)\tag{4}$$

$$77 \leq Cb \leq 127\ AND\ 133 \leq Cr \leq 173\tag{5}$$

$$(4)\ OR\ (5)\tag{6}$$

In (Vansh et al., 2020), an improved face detection approach was proposed using YCbCr color space and Viola-Jones algorithm. They state that detection time was a little longer than a simple Viola-Jones algorithm, however, the detection rate was increased. In this paper, we focus on the parallelization of the algorithm to improve the speed of detection and we add the skin color pre-treatment to increase the detection rate and minimizing the treated area in order to improve the speed. In the proposed method we add a treatment, after the integral image calculation, to select only the pixels that had skin color to be processed by the cascade classifier.


(a) (b)

(c)

Figure 6: Example of skin color segmentation.

For smaller images, since the face size may be less than (25x25), we used the nearest neighbor algorithm to scale up the image size before starting the previously described algorithm. We found that adding the scaling up can improve the detection rate by a factor of 1.27x on the small image.

An overview of our optimized CUDA implementation is presented in Figure 7.

5 EXPERIMENTATION AND DISCUSSION

5.1 Experimental Setup

The proposed method was developed and tested on Intel® Core^{TM} i5 2.30 GHz loaded with Windows 10 (64 bit) and NVIDIA graphics processing unit GeForce 920MX. The development and testing have been done in Microsoft Visual Studio 14.0.25431.01. The CUDA files are compiled by the CUDA compiler of Release 10.1, 10.1.168 with the architecture support corresponding to compute capability 5.0.

The detection time is related to five factors: the size of the image, the number of features in the classifiers, the step of scanning, the resizing scale and the hardware platform (Wei and Ming, 2011). In addition, the number of faces in the image affects the detection time. In this work, we maintain the classifier, the step of scanning, scaling factor and hardware plat-

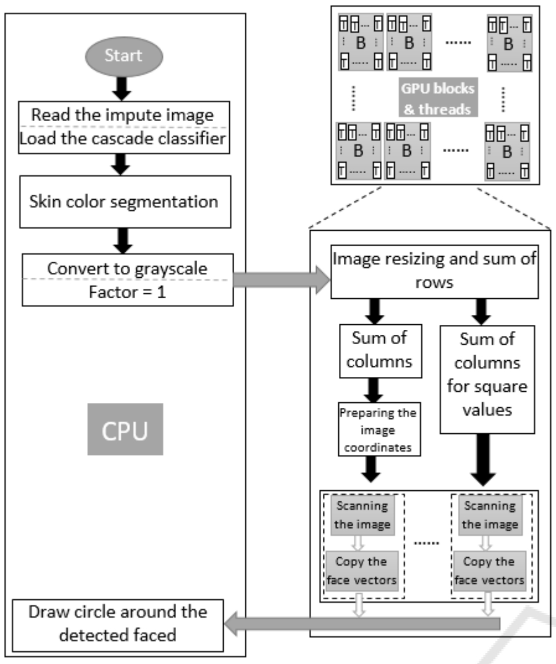

Start
Read the impute image
Load the cascade classifier
Skin color segmentation
Convert to grayscale
Factor = 1
CPU
Draw circle around the detected faced
GPU blocks & threads
B
Image resizing and sum of rows
Sum of columns
Sum of columns for square values
Preparing the image coordinates
Scanning the image
Copy the face vectors

Figure 7: Our optimized CUDA based face detection algorithm.

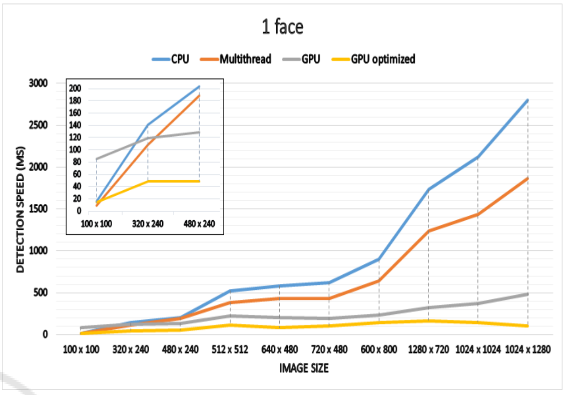
form unchanged, and compute the acceleration of our system. However, we were not able to find any publicly available databases contain color images with different sizes and a variant number of faces. So in order to evaluate the performance of the final face detection algorithm, the considered database is instead a collection of 700 images taken from the web. The collected database contains frontal face images in color with 10 different sizes (100x100, 320x240, 480x240, 512x512, 640x480, 720x480, 600x800, 1280x720, 1024x1024, 1024x1280), that are grouped by the number of faces, 7 different types were distinguished (1, 2, 3, 4, 5, 6-9, ≥10 faces), with 10 images for each size and for each number of faces.

5.2 Results and Discussion

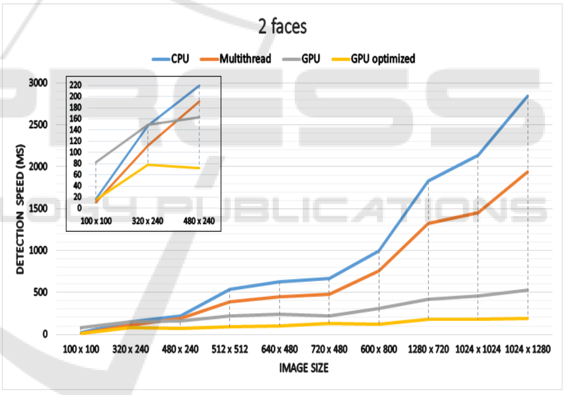
Our experiments are based on the comparison of our proposed solution and several implementations of Viola-Jones: the sequential one, multithreaded and Cuda GPU based. The results show that our solution decrease the detection time, increase the detection rate and eliminates the false positive rate. Figure 8 shows the graphical results of our face detection (all the results details are given in table 2). The previously presented database (section 5.1) was used for the testing of the detector. Each figure presents the execution time, according to the number of faces for the different algorithms (sequential, multithreaded, GPU and our optimized algorithm). At the top left of the figures, we zoomed the execution times of the small image sizes, in order to make them clearer.

According to the figures, we distinguish 2 different cases, the first for small size images and the second for big size images.

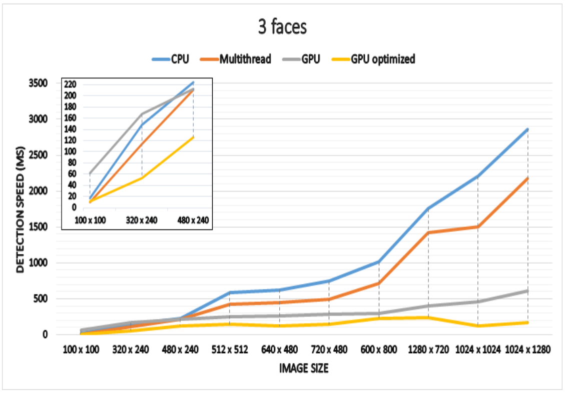
Figure 9 presents the detection time for the smaller size images. The Multithreaded version gives a better result than that of the CPU single-threaded and GPU.



(a)

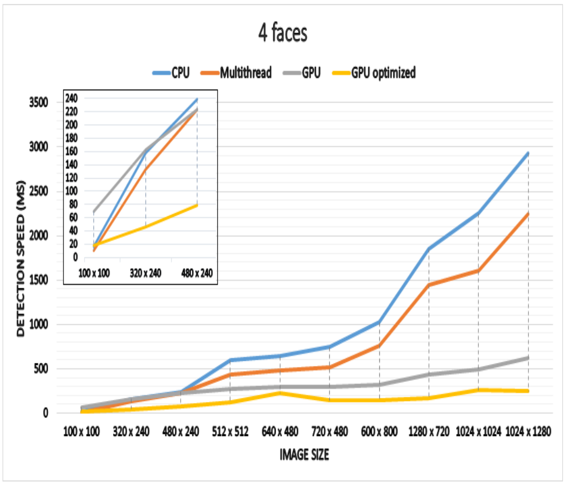


(b)

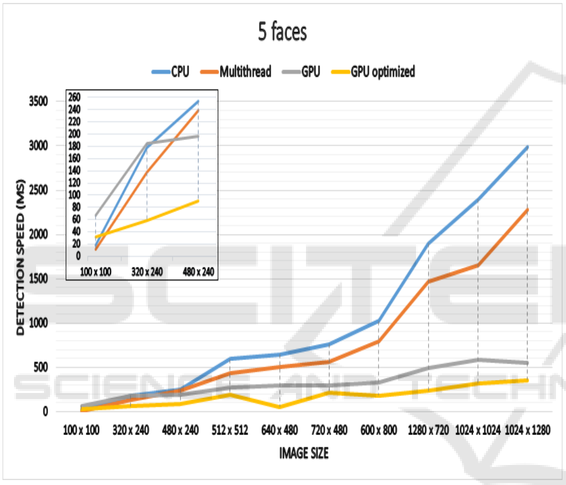


(c)

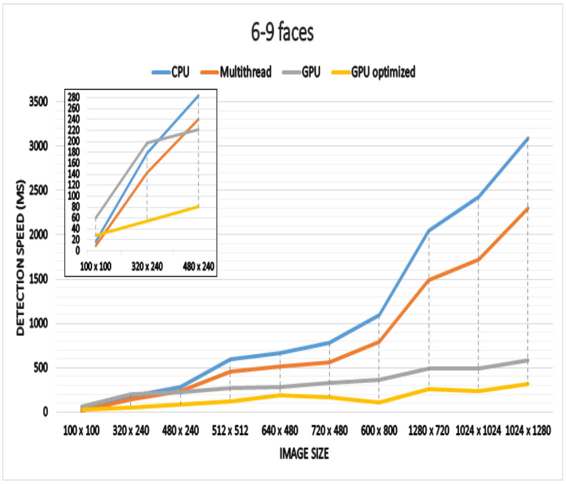
Figure 8: The detection time for all image sizes.



(d)

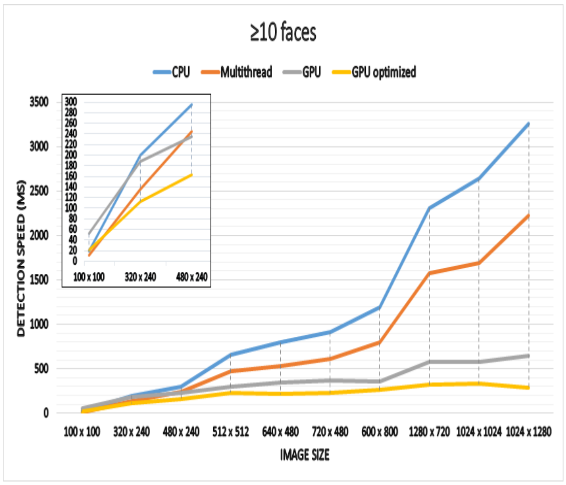


(e)



(f)

Figure 8: The detection time for all image sizes. (cont.).



(g)

Figure 8: The detection time for all image sizes. (cont.).

We notice that the GPU detection time increase for the small images. However, our optimized algorithm offers nearly the same execution time as the CPU version.

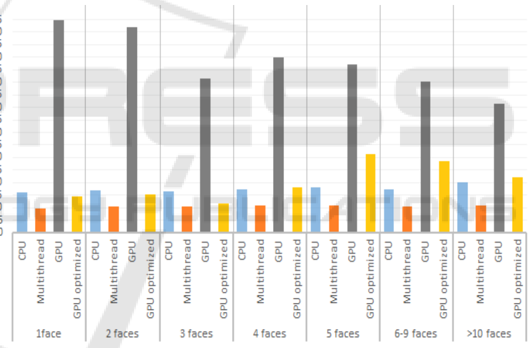


Figure 9: Detection time for small images (100x100).

For bigger size images, we notice that the GPU detection time is better than that of the CPU, and the space between their representation getting bigger with image size increases. From figures 8, it is obvious that our algorithm outperforms the CPU single-threaded and multithreaded versions and even the GPU version.

The performance of the proposed CUDA optimized algorithm with and without skin color filtering has been compared with that of the sequential and multithreaded implementation. Table 2 summarizes the detection time in ms for all the previously described implementations. As per the implemented results, it has been observed that the optimized GPU is better than the CPU version in most of the cases.

In the first experiment, we measured the processing time of sequential and multithreaded versions. we found that the multithreaded version can accelerate

Table 2: The detection time for the different implementations on the created dataset.

size		100 x 100	320 x 240	480 x 240	512 x 512	640 x 480	720 x 480	600 x 800	1280 x 720	1024 x 1024	1024 x 1280
1 face	CPU	16.20	140.93	203.55	523.77	581.76	620.68	899.05	1734.41	2119.42	2802.51
	Multithread	9.47	109.08	188.83	383.07	431.76	435.53	644.81	1236.44	1434.50	1860.77
	GPU	84.93	119.24	128.43	222.1	203.77	191.95	235.33	319.97	374.88	480.83
	GPU optimized	14.41	48.54	49.21	111.83	79.74	98.73	140.89	167.44	143.61	103.41
2 faces	CPU	16.70	147.69	219.18	542.80	623.4	663.11	998.03	1828.85	2140.54	2845.46
	Multithread	10.4	112.58	191.63	393.67	444.73	478.29	757.68	1324.35	1454.53	1940.97
	GPU	81.84	149.27	163.65	221.3	239.08	217.21	304.6	414.76	456.39	528.12
	GPU optimized	15.4	77.56	71.76	87.84	102.53	132.64	117.92	183.97	178.96	193.94
3 faces	CPU	16.65	148.94	224.54	586.58	625.12	744.6	1010	1754.41	2206.9	2863.15
	Multithread	10.41	114.32	210.80	428.55	446.43	495.34	710.22	1417.58	1508.11	2176.85
	GPU	61.54	167.57	212.71	251.96	259.69	283.91	301.24	400.17	457.7	610.65
	GPU optimized	11.65	53.19	125.94	143.14	118.09	141.26	227.39	238.65	126.52	171.61
4 faces	CPU	17.35	158.79	239.24	594.14	641.03	747.02	1022.28	1851.67	2255.01	2930.95
	Multithread	10.77	132.98	222.96	435.97	483.91	512.15	759.30	1440.53	1606.86	2242.37
	GPU	69.87	162.35	223.72	276.56	293.9	301.17	314.22	438.56	492.47	621.82
	GPU optimized	17.93	46.18	78.85	124.47	221.31	140.52	142.46	165.28	259	249.18
5 faces	CPU	17.99	178.03	254.61	595.34	646.07	762.86	1026.78	1901.63	2396.42	2987.78
	Multithread	11.01	137.97	239	438.07	510.40	562.95	789.87	1462.99	1657.34	2280.72
	GPU	67.27	185.1	196.11	267.15	295.64	299.49	333.74	491	581.57	547.61
	GPU optimized	31.33	59.06	89.98	197.4	55.99	219.45	184.51	233.79	316.75	354.17
6-9 faces	CPU	17.24	179.41	284.42	601.1	671.13	778.42	1092.92	2048.3	2426.02	3092.29
	Multithread	10.45	142.27	240.91	455.25	511.16	563.37	792.50	1494.26	1719.13	2304.84
	GPU	60.47	198.03	221.98	268.34	281.88	325.62	363.74	489.09	486.2	589.29
	GPU optimized	28.41	53.79	80.65	125.9	194.58	171.82	114.02	256.24	237.87	316.64
$\geq$10 faces	CPU	19.97	199.6	295.38	661.58	793.43	913.5	1195.36	2306.87	2637.04	3259.85
	Multithread	11.04	136.12	244.78	468.06	530.52	609.14	796.13	1577.59	1697.14	2223.2
	GPU	51.48	188.17	235.01	296.11	351.4	366.53	361.85	583.5	574.23	650.47
	GPU optimized	22	112.13	163.53	235.32	214.39	227.85	262.03	320.03	329.92	293.54

the execution by a factor up to 1.81x. However, the improvement isn't enough for realistic scenarios. We notice that the detection time is increasing with the rise of image size and the number of faces. After that, a GPU CUDA based version was implemented. For smaller images, we notice that the detection time was increased. Despite this, for the bigger sizes, the simple GPU implementation is faster than CPU implementations by a factor up to 5.83x. Since the GPU version still not that much faster and suffers when we use small images, an optimized version was implemented. For smaller images, the detection time becomes so near to that of the CPU versions. The improvement is 27.1x compared to the CPU version.

Although, it is important for a face detection algorithm to be fast, however, it is not the only factor to measure its efficiency. The detection rate and false positive are also important. First, for the GPU version, we achieved a detection rate of 91.35% with 15 false positives. In the GPU optimized version with the skin color segmentation, since only skin pixels are treated, the false positive is reduced to zero and the detection rate is 97.05%. With the use of stable parameters, it is obvious that our proposed implementation outperforms those of the literature. However, there is still a possibility of improvement, with a better cascade classifier and more amount of shared memory that exists in newer NVIDIA architecture.

6 CONCLUSIONS

Face detection is a classical computationally-intensive problem. In this paper, we solve this computationally intensive problem on an NVIDIA GPGPU using the CUDA parallel computing language. We propose a real-time optimized and robust GPGPU implementation of face detection algorithm using the RGB and YCbCr spaces of color to select the skin color pixel on which the detection is applied in order to reduce the research area. To evaluate our method, we created a database of 700 color images that contain different sizes and face numbers. We achieved an average of 27.1x acceleration over the CPU implementation. In order to increase the detection rate for the smaller images, we increased the size before treatment. This scaling-up increases the detection rate by a factor of 1.27x. With the use of skin color segmentation and small images scaling up, we could increase the detection rate to 97.05% and we get rid of the false positive. We believe that with a more robust classifier and more optimizations, we can still achieve a better speedup. Then, we will focus on the creation of a robust classifier and the optimization of the implementations to overcome the execution overhead caused by the number of faces. In addition, our challenge is to approve the efficiency of our approach regardless of the hardware platform used.

REFERENCES

Bhutekar, S. J. and Manjaramkar, A. K. (2014). Parallel face detection and recognition on gpu. *International Journal of Computer Science and Information Technologies*, 5(2):2013–2018.

Bilaniuk, O., Fazl-Ersi, E., Laganiere, R., Xu, C., Laroche, D., and Moulder, C. (2014). Fast lbp face detection on low-power simd architectures. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 616–622.

bin Abdul Rahman, N. A., Wei, K. C., and See, J. (2007). Rgb-h-cbcr skin colour model for human face detection. *Faculty of Information Technology, Multimedia University*, 4.

Chouchene, M., Sayadi, F. E., Bahri, H., Dubois, J., Miteran, J., and Atri, M. (2015). Optimized parallel implementation of face detection based on gpu component. *Microprocessors and Microsystems*, 39(6):393–404.

Devrari, K. and Kumar, K. V. (2011). Fast face detection using graphics processor. *International Journal of Computer Science and Information Technologies*, 2(3):1082–1086.

Fayez, M., Faheem, H., Katib, I., and Aljohani, N. R. (2016). Real-time image scanning framework using gpgpu-face detection case study. In *Proceedings of the International Conference on Image Processing, Computer Vision, and Pattern Recognition (IPCV)*, page 147. The Steering Committee of The World Congress in Computer Science, Computer ....

Hefenbrock, D., Oberg, J., Thanh, N. T. N., Kastner, R., and Baden, S. B. (2010). Accelerating viola-jones face detection to fpga-level using gpus. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 11–18. IEEE.

Jain, V. and Patel, D. (2016). A gpu based implementation of robust face detection system. *Procedia Computer Science*, 87:156–163.

Jeong, J.-c., Shin, H.-c., and Cho, J.-i. (2012). Gpu-based real-time face detector. In *2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 173–175. IEEE.

Jiang, N. and Wang, L. (2015). Quantum image scaling using nearest neighbor interpolation. *Quantum Information Processing*, 14(5):1559–1571.

Khan, M. A., Shaikh, M. K., bin Mazhar, S. A., Mehboob, K., et al. (2017). Comparative analysis for a real time face recognition system using raspberry pi. In *2017 IEEE 4th International Conference on Smart Instrumentation, Measurement and Application (ICSIMA)*, pages 1–4. IEEE.

Kong, J. and Deng, Y. (2010). Gpu accelerated face detection. In *2010 International Conference on Intelligent Control and Information Processing*, pages 584–588. IEEE.

Li, E., Wang, B., Yang, L., Peng, Y.-t., Du, Y., Zhang, Y., and Chiu, Y.-J. (2012). Gpu and cpu cooperative accelaration for face detection on modern processors. In *2012 IEEE International Conference on Multimedia and Expo*, pages 769–775. IEEE.

Meng, R., Shengbing, Z., Yi, L., and Meng, Z. (2014). Cuda-based real-time face recognition system. In *2014 Fourth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 237–241. IEEE.

MRF (2019). Biometric in government market research report - global forecast til 2025. page 195. MARKET RESEARCH FUTURE. https://www.marketresearchfuture.com/reports/biometrics-government-market-8035.

Mutneja, V. and Singh, S. (2018). Gpu accelerated face detection from low resolution surveillance videos using motion and skin color segmentation. *Optik*, 157:1155–1165.

Mutneja, V. and Singh, S. (2019). Modified viola–jones algorithm with gpu accelerated training and parallelized skin color filtering-based face detection. *Journal of Real-Time Image Processing*, 16(5):1573–1593.

Nguyen, T., Hefenbrock, D., Oberg, J., Kastner, R., and Baden, S. (2013). A software-based dynamic-warp scheduling approach for load-balancing the viola–jones face detection algorithm on gpus. *Journal of Parallel and Distributed Computing*, 73(5):677–685.

OpenCV. [Online]. https://sourceforge.net/projects/opencvlibrary/.

Oro, D., Fernández, C., Saeta, J. R., Martorell, X., and Hernando, J. (2011). Real-time gpu-based face detection in hd video sequences. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 530–537. IEEE.

Oro, D., Fern'ndez, C., Segura, C., Martorell, X., and Hernando, J. (2012). Accelerating boosting-based face detection on gpus. In *2012 41st International Conference on Parallel Processing*, pages 309–318. IEEE.

Patidar, S., Singh, U., Patidar, A., Munsoori, R. A., and Patidar, J. (2020). Comparative study on face detection by gpu, cpu and opencv. *Lecture Notes on Data Engineering and Communications Technologies*, 44:686–696.

Saikia, P., Janam, G., and Kathing, M. (2012). Face detection using skin colour model and distance between eyes. *International Journal of Computing, Communications and Networking*, 1(3).

Shaik, K. B., Ganesan, P., Kalist, V., Sathish, B., and Jenitha, J. M. M. (2015). Comparative study of skin color detection and segmentation in hsv and ycbcr color space. *Procedia Computer Science*, 57(12):41–48.

Sharma, B., Thota, R., Vydyanathan, N., and Kale, A. (2009). Towards a robust, real-time face processing system using cuda-enabled gpus. In *2009 International Conference on High Performance Computing (HiPC)*, pages 368–377. IEEE.

Shifa, A., Imtiaz, M. B., Asghar, M. N., and Fleury, M. (2020). Skin detection and lightweight encryption for privacy protection in real-time surveillance applications. *Image and Vision Computing*, 94:103859.

Sun, L.-c., Zhang, S.-b., Cheng, X.-t., and Zhang, M. (2013). Acceleration algorithm for cuda-based face detection. In *2013 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC 2013)*, pages 1–5. IEEE.

Tek, S. C. and Gökmen, M. (2012). Gpu accelerated real-time object detection on high resolution videos using modified census transform. In *VISAPP International Conference on Computer Vision Theory and Applications*, pages 685–688.

Vansh, V., Chandrasekhar, K., Anil, C., and Sahu, S. S. (2020). Improved face detection using ycbcr and adaboost. In *Computational Intelligence in Data Mining*, pages 689–699. Springer.

Viola, P. and Jones, M. (2001). Robust real-time face detection. In *null*, page 747. IEEE.

Wai, A. W. Y., Tahir, S. M., and Chang, Y. C. (2015). Gpu acceleration of real time viola-jones face detection. In *2015 IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*, pages 183–188. IEEE.

Wei, G. and Ming, C. (2011). The face detection system based on gpu+ cpu desktop cluster. In *Intl. Conf. on Multimedia Technol.(ICMT'11)*, pages 3735–3738.

Zafeiriou, S., Zhang, C., and Zhang, Z. (2015). A survey on face detection in the wild: past, present and future. *Computer Vision and Image Understanding*, 138:1–24.