GPU-accelerated High-speed Eye Pupil Tracking System

Juan Mompeán*†, Juan L. Aragón*, Pedro Prieto† and Pablo Artal†
*Dept. de Ingeniería y Tecnología de Computadores
†Laboratorio de Óptica, IUiOyN
Universidad de Murcia
{juan.mompean, jlaragon, pegrito, pablo}@um.es

***Abstract*—Pupil tracking under infrared illumination is an important tool for many researchers in physiological visual optics and ophthalmology. It is also a relevant topic for gaze tracking which is used in psychological and medical research, marketing, human-computer interaction, virtual reality and other areas. A typical setup can be either a low-cost webcam with some infrared LEDs or glasses with mounted cameras and infrared illumination. In this work, we evaluate and parallelize several pupil tracking algorithms with the aim of estimating the pupil's position and size with high accuracy in order to develop a high-speed pupil tracking system. To achieve high processing speed the original non-parallel algorithms have been parallelized by using CUDA and OpenMP. Graphics cards are designed to process images at very high frequencies and resolutions, and CUDA enables them to be used for general purpose computing. Our experimental results show that pupil tracking can be efficiently performed at high speeds with high-resolution images (up to 530 Hz with images of 1280x1024 pixels) using a state-of-the-art GP-GPU.**

I. Introduction

Pupil tracking is an important approach to know key information about the eye and also for supporting a lot of different applications on both the research and commercial arena, specially when used for gaze tracking. There are many application cases of the later, being an interesting one the remote control of computers with the subject's gaze: it can be used instead of the mouse, for moving the camera in 3D virtual reality scenarios. But it is also a powerful tool for disabled people with very reduced movement capabilities who can control a computer with their gaze.

Pupil tracking is very often performed by using infrared illumination because it does not perturb the subject, neither annoys him nor it affects the eye's properties since the subject cannot see the infrared light. In particular, the pupil size does not change with infrared illumination, which could be useful in some experiments. Pupil tracking under infrared illumination is commonly used in visual optics experiments and in ophthalmic instruments. In some cases it is a key component of the system because the actions that are performed need to be done in the center of the pupil, which in turn needs to be calculated with high accuracy and at a high speed. While for eye gazing a low precision on the pupil determination does not affect much the results [1], in some visual optics applications a low precision can lead to important errors in the final results.

Adaptive Optics Visual Simulation [2], consisting of the manipulation of ocular aberrations in order to perform visual testing through a modified optics, is an example of application that would benefit from a fast and accurate pupil tracking procedure. Nowadays, visual simulation systems rely on the subject's pupil being steady with respect to the aberration generator. This is usually achieved by means of a bite bar or a chin rest, neither element comfortable nor able to completely fix the subject's head. In this context, a fast and accurate pupil tracking system would be a requirement for a free-floating visual simulator where pupil movements are detected and compensated, e.g., with galvanometric mirrors.

There are several algorithms for calculating the pupil position and size, in some cases tailored for a particular optical setup and illumination. In this paper, we have selected three of the most common (and non-invasive) algorithms that can be found in the literature. One promising and robust algorithm is called Starburst [3], which is based on tracing a number of rays that search big gradient changes and it is capable of providing information not only about pupil size and position but also its shape and orientation. A second and more standard algorithm applies a threshold to the image and labels it for iteratively searching the biggest labelled circle [4]. The third evaluated approach uses the resulting edges after applying a Canny edge detector [5] to finally apply the Hough transform [6] in order to calculate the pupil's position and size.

In this paper we parallelize and evaluate the aforementioned pupil tracking algorithms in different computing systems (from general purpose multicore CPUs to state-of-the-art GP-GPUs) by using both OpenMP [7] and CUDA [8] programming environments, with the final goal of designing a high performance (up to 85.5x speedup compared to their corresponding sequential implementations) that allows for high speed processing (up to 530 frames per second) while not sacrificing a high detection accuracy (more than 90% of the pupils in the dataset are detected with an error in their radio lower than 5%).

It is important to note that despite the fact that pupil tracking is a widely used tool, it is still hard to achieve high speed pupil tracking with high quality images. Accurately tracking the pupil at high speed is a computationally intensive task and, commonly, images require some pre-processing treatment which is also computationally intensive, making it challenging to process high-definition images, or even tracking both eyes simultaneously. In this sense, the three evaluated pupil tracking algorithms offer a different pupil detection accuracy depending on the input image and they also present big differences in

the computational requirements to perform their calculations, as we will show in Section VI. For that reason, a second contribution of this paper is a deep and exhaustive evaluation of the selected algorithms by covering a wide range of tuning parameters for each particular case.

II. Background and Related Work

Different algorithms have been developed to track the pupil position and size under different conditions. A robust and well known approach is the Starburst algorithm [3]. This is based on a series of rays that extend from an initial position to the edges of the image. A change over a threshold is searched in the intensity derivative of the pixels in each of the rays. Then new rays are created, that go from the newly found points to the initial position. By doing this, points on the other side of the pupil are found. Since only the pixels that belong to the rays are analyzed, a small amount of pixels of the image are accessed, which reduces the computation time. While it is an iterative algorithm, experimental results have shown that in most cases it converges after two or three iterations.

Another approach is to use a Gradient Vector Flow (GVF) snake [9]. In this algorithm an estimate of the pupil center is calculated to reduce the area where the pupil is searched. In order to estimate the pupil's position only the middle sub image is taken into account, which is usually beneficial since the pupil is commonly located in the center of the image in most controlled experimental setups. However, this is not always the case and this approach will fail to detect pupils in some experiments or in non-ideal environments. Still the algorithm is interesting because it is able to find the pupil with high accuracy when the initial assumptions are true.

Pupil detection can also be performed with a technique called Graph Cuts [10]. In the proposed algorithm the pixels constitute the nodes of a graph while the edges represent their relationship. So the Graph Cut algorithm tries to find the minimum *cut* between the pupil and the rest of the image. As a previous step, an eyelash detection algorithm is used to remove them. This is a basic pre-processing stage in order to correctly select a pixel from inside the pupil and a pixel from the rest of the image, which is important because the pixel from the pupil is selected as a dark pixel in the image and the algorithm could select a pixel from the eyelashes instead, provoking the algorithm to fail.

Another simpler approach is to use the Hough Transform on the edges detected with the Canny algorithm [11]. This method is also used by Masek [12] with a specific pre-processing for removing the eyelids. Although it was initially proposed for segmenting the iris, it can be used for pupil tracking as well. It is a simple approach, however, its effectiveness and accuracy are very dependent of the input set of pictures. In addition, this algorithm is very sensitive to noise added by eyelashes or even other smaller features of the eye. Furthermore, it is a very slow approach when used in large images with a wide range of pupil's radios since it will search sequentially for all the targeted pupil sizes in all the pixels of the image.

Blobs have also been applied [13] by using the central blob of the image as the pupil. This is a simple technique because by using a threshold the pupil can easily be isolated from other features of the eye. However, eyelids and eyelashes could be overlapping and they could be detected as part of the pupil. To avoid such errors eyelids and eyelashes detection mechanisms should be previously performed which increases the computational demands of the algorithm.

GPU processors have been used for pupil tracking with color images [14]. Borovikov proposed a custom blob detector; he assumes that the pupil will be near the center of the image and searches iteratively a dark blob. In each iteration the center of mass of a circle around the current estimated center is calculated providing higher values to the darker pixels. After each iteration the initial radio is decreased and the algorithm stops when the center position does not change between two consecutive iterations. Another GPU approach for pupil tracking was proposed by Mulligan in [15]. It is based on searching an area within a threshold in the image. This approach may have problems to find the pupil when the eyelashes are partially hiding it or when there are corneal reflections inside the pupil. Under good experimental conditions it achieves a processing speed of up to 250 Hz but for low resolution images.

Another field closely related to pupil tracking is gaze tracking. In the literature there are a number of approaches to perform it [16]. Usually, a wide-field angle camera with infrared illumination is used to allow the user to freely move its head. As a trade-off, the resolution for detecting the eye is highly reduced as well as the precision for detecting the pupil [17], [18]. Although it is possible to focus closer to the eye and move the camera to follow the subject it is a complex and expensive setup that fails if the subject moves fast. An alternative consists of using a head-mounted system that allows head movements [19]. Another approach uses the dark and bright pupil for tracking the eye [20], [1] which simplifies the pupil detection but increases the complexity and cost of the setup since it must synchronize the camera with the two rings of LEDs.

III. GPU-based Real Time Pupil Tracking

The main goal of this work is to parallelize several state-of-the-art pupil detection algorithms to develop a high speed pupil tracking system capable of processing high quality images with an accurate pupil determination. A deep exploration and a throughout evaluation must be carried out, by varying the input parameters of these algorithms to determine how they behave with different tuning values and, more specifically, to characterize their run-time performance and the achieved accuracy, in terms of relative error compared to the correct pupil size and position.

A. Preprocessing

In order to perform a fair comparison of the algorithms, the same preprocessing is applied to all the images. The set of images used for the experiments (refer to Section IV for additional details) are sharp enough and they do not have significant noise. However, as it will be also explained in Section IV, the illumination setup added corneal reflections. Every image taken with direct infrared illumination will have similar reflections, so an algorithm for removing them is used.

The algorithm used for this purpose was proposed by Aydi *et al.* [21]. It is a simple method that applies a top-hat transform, followed by an image thresholding, an image

dilation and, finally, an interpolation of the detected points. An example of the results of the corneal reflection removal step is shown in Figure 2b, which can be compared with the original image in Figure 2a. The dilation and the erosion filters are computationally intensive, so a naïve implementation in CUDA is not suitable for high performance applications. However, these are separable filters which can be expressed as the outer product of two vectors. The thresholding filter is a straightforward operation in CUDA, the values of the pixels above the threshold are interpolated using the values of the pixels in the border of the corneal reflection. By weighting the pixel values based on their distance to the replaced pixel, a smooth result is generated avoiding the creation of new borders inside the pupil.

After the corneal reflection removal step, a Gaussian filter is applied (with a kernel size of 5 and a sigma of 2). The naïve implementation of the Gaussian filter in CUDA is straightforward, however, it is not very efficient. Similarly to the approach used with the erosion and dilation two separable kernels are used. These CUDA optimizations are further described in Section V. A median filter has been also tested as a replacement for the Gaussian filter. The median filter has the advantage of removing small features while keeping the pupil edges: it selects the median value of the pixels around each pixel with a chosen radio. To implement a parallel median filter, an histogram is used. Each thread calculates the histogram of the pixels in the area around its center and scans the histogram to find the element with half of the values on each side. A texture has been used to optimize the memory accesses of the GPU threads.

B. Threshold and Labelling Algorithm

A similar approach to the proposed by Rankin *et al.* [13] is used. Although instead of selecting the central blob we have modified the original algorithm to select the biggest blob. For selecting the threshold several values are tested using an iterative algorithm which tests values ranging from 20 to 100. Depending on the step size of the iterative algorithm the accuracy will be higher or lower and also the time spent in the algorithm is linearly related to this step size (in our experiments the chosen step size varies from 5 to 40). Therefore, the performance and the accuracy achieved by this algorithm are a trade off, the faster the less accurate, and vice versa. After applying the threshold to the image, as described before, the labelling process is performed. A state-of-the-art labelling algorithm by Chen [22] has been used. In the labelling algorithm the image is scanned to find the connected areas and to assign a different label to each of them.

After labelling the image, the histogram of the image is calculated. By calculating the histogram of the labelled image the biggest labelled area can easily be found by searching the maximum value within the histogram. Once the maximum label is found the image is scanned to remove the rest of labels while keeping the biggest labelled area. Then the image is scanned for searching the border of the labelled area. A simple algorithm for locating borders is used, which searches empty pixels in the image and checks if any of its eight neighbours is different than 0. If any of them is different its position is stored in a list of border pixels.

With the list of found border pixels a RANSAC (Random Sample Consensus) circle fit is performed [23] to determine the pupil. The RANSAC algorithm fits very well the typical output of this algorithm. Usually, there are a lot of points found in the pupil border in addition to some points outside the pupil. If a normal circle fit were performed for all the points detected, the outliers would distort the result. While the RANSAC algorithm uses only a few points to fit a circle and then checks how many of the points are closer than a given distance to the fitted circle (number of votes). We have used 5 points to do the circle fit and a maximum distance of 2 pixels to accept the vote of a point. The initial set of 5 points is randomly selected. Generating random numbers is a slow operation but they can be generated only once and then be re-used for all of the analyzed images, as described in the CUDA optimization Section V. Every time that RANSAC is performed 1024 fits are calculated, using 1024 sets of 5 points. To generate the 1024 sets of 5 points the random numbers are multiplied by the total amount of points found, generating indexes in the array of points. Then a circle fit is performed for each set of points using the circle fit developed by Taubin [24]. Afterwards, the votes are computed and a reduction is performed to add the votes of each circle. Finally, the circle with the maximum number of votes is selected as the best fit. The process is repeated for all the thresholds and the circle with most votes is selected as the circle describing the pupil.

C. Starburst Algorithm

The second algorithm is called Starburst, developed by Dongheng and Parkhurst [3]. It is based on the fact that the pupil border is usually the place with the biggest gradient values. This algorithm is very sensitive to the corneal reflections, so the preprocessing step for removing them is crucial.

After removing the reflections the Starburst algorithm is used to search the pupil. Starburst needs a starting central point of the pupil. When no previous information is available the center of the image is typically selected. When previous information is available, for example in a live capture, using the previous center can significantly speed-up the process and also increase the accuracy in some cases. A variable number of rays, we have used 20, are "launched" from the center to the limits of the image. Each ray is projected in a different direction, dividing a circle in equal portions. The ray uses the pixels that are in its direction to calculate its gradient which is later inspected to search the first value bigger than a threshold. The result of this stage are the points found in the rays.

At the beginning the rays' directions are calculated, and each direction is calculated by a CUDA thread. Then, for each ray another CUDA thread calculates the gradient along the ray. If the pixel value is over the threshold its position in the ray is stored using the *atomicMin* CUDA operation. Thereby, the closest point to the origin of the ray is saved. Finally, the position in the image of each pixel found on each ray is calculated by using its position in the ray, the center and the direction of each ray.

In the second stage of the Starburst algorithm those points are used to create new rays from them to the center. The new rays are limited to an angle of $\pm 50^{o}$ instead of the 360^{o} used initially. The purpose is to find points on the other side of

the pupil and avoid going out of it. The same method for parallelizing the process in CUDA is used in this second stage.

Finally, the points from both stages are joined in the same array and a RANSAC circle fit is performed, as described for the previous algorithm. In the original Starburst algorithm the average location of the points was used, in our implementation we utilize the RANSAC circle fit since it is a faster approach and it can speed-up the convergence. The distance between the new center and the old center is calculated and, if it is smaller than 10 pixels, the algorithm finishes. Otherwise, another iteration is executed until it converges or ten iterations have been performed.

D. Canny Edge Detector and Hough Transform Algorithm

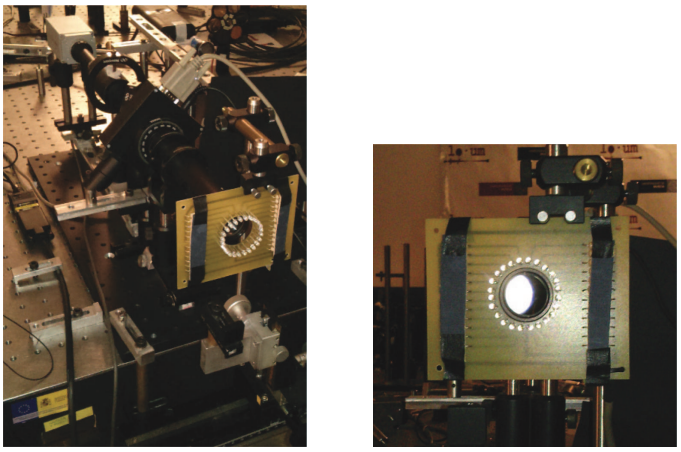
The third evaluated algorithm uses the Canny edge detector [5] and the Hough Transform [6]. For performing the Canny edge detection, the image is smoothed horizontally and vertically with a 1-D Gauss kernel and with the first derivative of a Gauss kernel. Four images are generated as the result of the process. Then the gradient value and direction are calculated using a naïve CUDA implementation.

The value of the *high threshold* of the Canny edge detection is calculated as the minimum value which is bigger than the 80% of the pixels' values. The histogram of the images is used to calculate this high threshold. On the other hand, the *low threshold* is established as 40% of the *high threshold*. The maximum value is calculated first in shared memory for each block and then in global memory for all the blocks. The gradient is normalized with the calculated maximum value using a naïve CUDA implementation. The non-maximum suppression step is performed using the normalized gradient and the gradient direction matrices. Then the double thresholding is performed and finally the hysteresis is applied to remove the low values and keep the high ones.

With the result of the Canny edge detector the Hough Transform is applied. The Hough Transform will check for each pixel every possible radio in the range selected. So one thread is issued for every pixel in the image for each one of the checked radios. Each thread counts the number of pixels that are in the circumference with center in its position and with its radio and belong to an edge. An *atomicMax* CUDA function is used to check if the current circle has more votes than the circles previously found. If it is the case, its center and radio is stored. The final result is the circle which obtains the maximum number of votes.

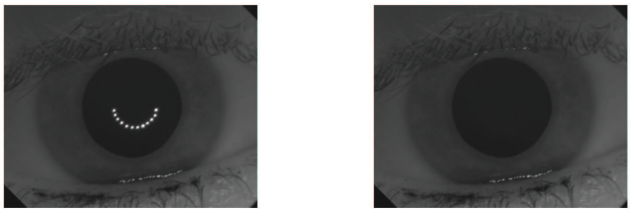
IV. Experimental Framework

While many different setups can be used for performing pupil tracking, our approach consists of a set of infrared LEDs and a camera focusing at the eye (Figures 1a and 1b). The setup appears more complicated because there are other optical elements since this setup was used in an experiment for locating the achromatizing pupil position and first Purkinje reflection in a normal population [25]. The camera is at the end of the setup, whereas the ring of LEDs is closer to the eye of the subject. Only the top part of the ring was used to avoid adding a circle to the images. In order to track the pupil the subject sits in front of the camera and stays there while the tracking is running. The tracking can be done in real-time



(a) (b)

Fig. 1: (a) Setup for pupil tracking. (b) Sub-system for infrared illumination.



(a) (b)

Fig. 2: (a) Eye image of a subject with the described setup. (b) Eye image of a subject with the corneal reflections removed.

at the same time that the subject is standing in front of the camera or later if a video is captured. An image of the eye of a subject captured with this setup is shown in the figure 2a. In that image it can be seen that the semicircular set of LEDs is generating corneal reflections that appear as bright points inside the pupil. A different distribution of the LEDs could produce better images, e.g., having only one powerful LED would be better since there would be only one spot. Having a semicircular ring can hide half of the pupil border in some subjects with very small pupils. Depending on the final purpose of the pupil tracking these subjects might be avoided or not, so the infrared illumination will have to be chosen taking this into consideration.

The GPU used for testing the CUDA programs is a high-end NVIDIA GeForce GTX 980, with 2048 CUDA cores and 4GB of memory. This GPU has one of the biggest amount of cores currently available. That enables our programs to exploit all the parallelism in the algorithms. For the CPU and OpenMP experiments an Intel i7-2600K processor with 4 physical cores and hyperthreading has been used.

Pupil Images Dataset. The experiments were performed using 964 pictures of 1280x1024 pixels taken with the described optical setup from 51 different subjects. A subset of this image collection was used in [25]. The dataset contains pupils with a wide variety of radios, ranging from $1.66mm$ to $4.28mm$, with

a mean of $3.21mm$. In addition, the evaluated set of images have different illumination intensities and include pupils with a high variety of shapes since they belong to different subjects and every pupil has a different shape.

V. Accelerating Pupil Tracking

CUDA enables developers to use the huge computing power of modern GPUs, but it does not come for free. While a naïve algorithm implementation in CUDA usually provides an important speedup, in order to achieve higher speedups some additional optimizations must be performed. The following paragraphs explain the different CUDA optimizations that have been used for the pupil tracking algorithms. Note that while some optimizations apply to all of the algorithms others only apply to Starburst. Figure 3 summarizes the speedup obtained by the Starburst algorithm for each CUDA optimization (and also an OpenMP implementation running in a multicore CPU) over a sequential implementation in a high-end CPU. The CUDA optimizations are cumulative and while each of them separately would not make a big performance impact, together, they are able to significantly reduce the global execution time.

Naïve CUDA Implementation. A naïve CUDA implementation directly maps the C code into CUDA. While the code is not optimized some good practices are followed: unnecessary memory copies are avoided, intrinsic functions are preferred (high precision is not necessary) and optimal block sizes are selected. The naïve implementation achieves a tracking speed of 38.6 frames per second and a speedup of 6.3x.

Pre-calculating random numbers. Random numbers are used to select the points used in the RANSAC method. They are generated with the *cuRAND* library [26] and then are scaled to match the range of the number of points previously found. Generating random numbers is an expensive task, so it should be avoided whenever possible. In the naïve implementation, the numbers are generated for each iteration of the Starburst, but that is inefficient. Since the points found will be different for each image and for each iteration the same random numbers can be used for all the images. It is still necessary to scale the numbers for each set of points to generate random numbers between 0 and $numberofpoints - 1$. After removing the generation of random numbers off the main loop and pre-calculating them at the beginning, the tracking speed is increased to 62.4 frames per second and the speedup comparing with the previous version is 1.6x (or 10x respect to the CPU).

Separable preprocessing kernels. The erosion, dilation and gaussian filter operations are performed with a *k*k* filter in the naïve implementation. As a result, they generate a lot of memory loads saturating the memory system while not fully utilizing the CUDA cores. Although new CUDA cards have big caches that partially mitigate the problem it is still possible to highly reduce the necessary bandwidth by dividing the kernel in two 1-dimensional kernels. The three operations can be divided in two kernels reducing the memory loads of each thread from *k*k* to *2*k*. And only an intermediate buffer, which can be pre-allocated, is needed. After adding the separable kernels the frames per second are increased dramatically up to 203.2 and a huge speedup of 32.7x is obtained.

Preprocessing kernels with shared memory. Although the memory bandwidth used by the preprocessing operations has been greatly reduced with the *separable kernels* optimization there is still room for improvement. Most of the pixels loaded by each thread are also loaded by others threads in the same block, so it is a good idea to preload the data once to shared memory [27] and do the calculations accessing the much faster shared memory. After adding the shared memory to the three preprocessing operations the frames per second are increased to 257.9 and the speedup over a CPU raises to 41.6x.

Preallocate memory. Allocating the memory in the GPU is relatively expensive, so allocating memory on a loop should be avoided. In the naïve implementation the memory is allocated for each image or even for each iteration of the Starburst for some variables that may slightly change its size through iterations. Since the maximum size of each dynamically allocated variable is known, they can be preallocated at the beginning and reused through the whole execution. After applying this optimization the obtained number of frames per second is 378.8 and the speedup increases to 61.1x.

Pinned memory. Copying the memory from the CPU to the GPU is an expensive task. Since the amount of time spent processing an image after applying the previous optimizations is very low, the overhead due to the memory copies is increasing its weight. It takes over 25% of the processing time to copy the images to the GPU. Each image is only copied once, so it cannot be reduced. However, it is possible to reduce the memory copy time by using pinned memory. After switching to pinned memory the number of frames per second obtained is 428.6 and the speedup over a CPU raises to 69.1x.

Reduction with shuffle instructions. The RANSAC is a key part of the Starburst algorithm, in order to find the best fitting circle it needs to sum the votes from every point from all the circles. It is a reduction which is performed in parallel for over 100 points for the 1024 circles. The naïve approach with an `atomicAdd` is very slow, in the naïve implementation we are using a typical approach with shared memory [28]. However, using *shuffle instructions* provides a big improvement, although the total time spent doing the reduction is low and the achieved speedup is modest. The new number of frames per second raises to 449.4 and the resulting speedup increases to 72.4x.

Join kernels While separating the functionality may simplify the code, it can lead to reduced performance. There are three functions where separating the functionality has resulted in repeated accesses to memory and, as a consequence, wasted computing time. These three functions are: the last part of the top-hat functions (second step dilation and subtraction) and the thresholding; generating gradients and searching the points in Starburst; and the RANSAC with the reduction. All of these functions shared the same pattern, they were accessing the data generated from the previous function. After joining them they are avoiding storing data to global memory that would be later loaded. After the optimization 512.6 frames per second and a speedup of 82.6x are achieved.

Template unroll. CUDA is capable of using C++ templates, which are useful for reusing the same functions with different types but are also useful for compiling a function several times for different values of a variable. After adding templates to the gaussian, the erosion and the dilation filters, the radio of these convolutions has been parameterized enabling the compiler to unroll the loop for each of these functions. To do so a

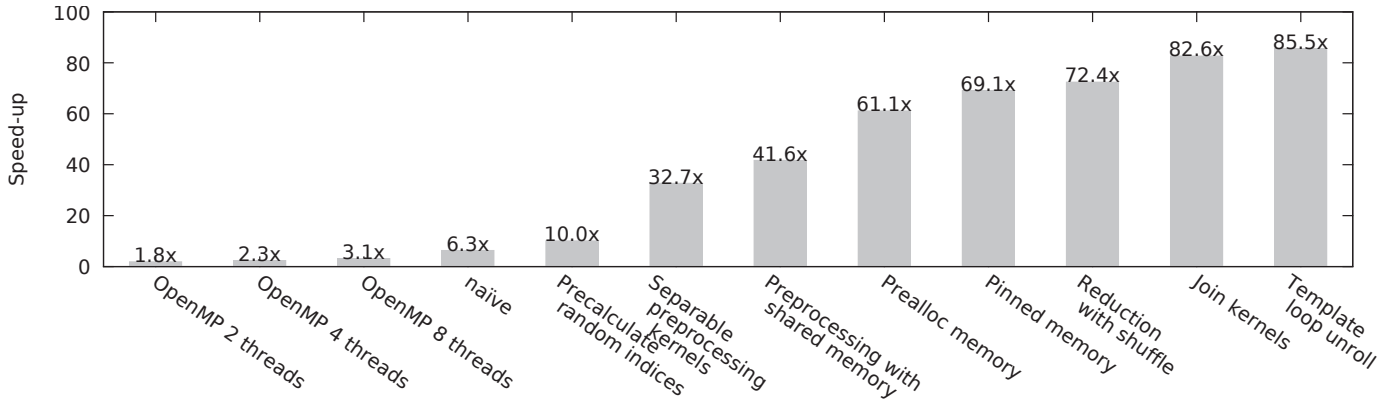


Fig. 3: Speedup of the parallelized StarBurst using OpenMP and the CUDA optimizations (over a sequential version).

`switch` with different radio sizes has been added to keep a small amount of flexibility in the radio size. Unrolling the loops avoid the tests that must be performed in each iteration of a loop to check if it must stop or not, therefore, reducing the execution time. The final number of frames per second raises to 530.3 and the final achieved speedup is 85.5x.

VI. Experimental Results Evaluation

A. Starburst Results

Figure 4a shows the accuracy of the Starburst algorithm applying six different image preprocessing configurations. Two of them use a Gaussian filter with a kernel size of 5 and a sigma of 2. Whereas four of them use a median filter with a kernel size of 7 or 11. The size of the corneal reflections removal kernel is also specified with values between 13 and 25.

The accuracy is measured as the percentage of images where the center and radio have been found within a given margin of error with respect to the correct position and size. E.g., if the error in the calculated radio is $0.1mm$, whereas the error in the center is $0.05mm$, and assuming a pupil radio of $2mm$ the relative margin of error are 5% and 2.5% for the pupil radio and the center, respectively. The biggest error is used, in this example a 5% corresponding to the pupil radio.

In Figure 4a it can be observed that there are some configurations with more than 85% of the images detected with an error under 5% and more than 95% detected with an error smaller than 10%. For the average pupil, which has a radio of $3.21mm$, a relative error of 5% would mean an absolute error of $0.16mm$ in the radio. But the error of the radio is on average 1% less than the error of the center: the same amount of pupils are detected within a margin of error of 5% for the center and within a margin of error of 4% for the radio. A relative error of 4% for the average radio would be an absolute error of $0.13mm$, and a relative error of 3% would be an absolute error of $0.096mm$.

In addition to that, a performance comparison is shown in Figure 4b. Clearly, the configurations using the median filter are slower than the configurations using the Gaussian filter due to the different performance of both methods.

B. Thresholding and Labelling Results

The accuracy of the Thresholding and Labelling algorithm (referred as T&L in Figure 5a) is quite good taking into account that this algorithm is very sensitive to overlappings of the eyelids and eyelashes. It is able to achieve an accuracy over 90% for small steps within a reasonable margin of error. Still its accuracy is decreased as the size of the steps is increased because it misses some good thresholds. The performance (Figure 5b) is directly related to the size of the steps used. Increasing the size of the steps decreases the execution time because it reduces the number of iterations of the algorithm. Relatively high speeds are achieved with step of size 20 and 40. After examining the accuracy and the performance of the algorithm it is clear that there is a trade-off between both. Increasing one of them will decrease the other, so a balance must be found.

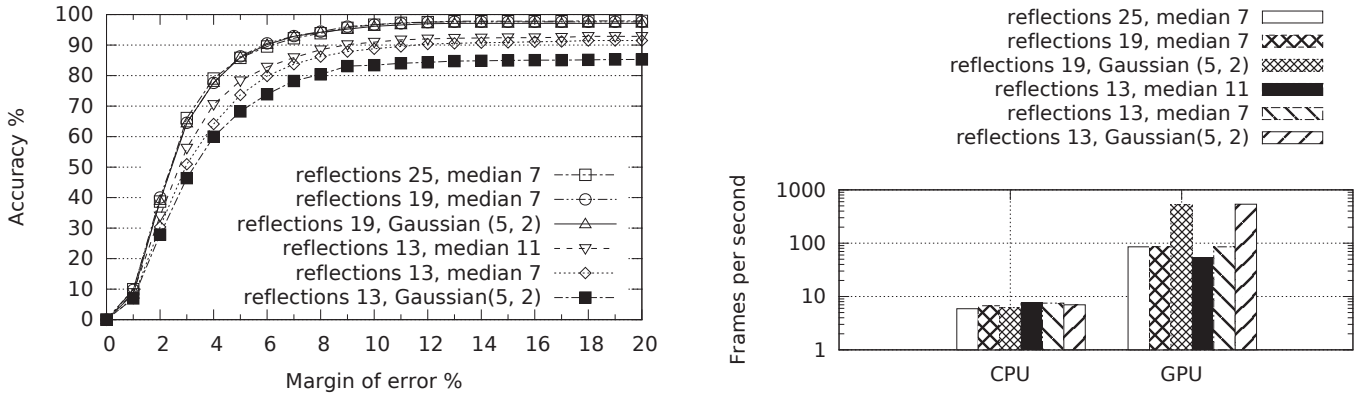
C. Hough Results

The Hough Transform turns out to obtain very accurate results (Figure 5a). It is able to detect the pupil correctly despite the iris, eyelashes and eyelids adding edges to the image. However, it is a very slow method (Figure 5b) since it is only capable of processing 4.7 frames per second in the GPU. The Hough Transform requires a huge computing power because it is checking all the possible radios for all the pixels.

D. Side-by-side comparison of the three approaches

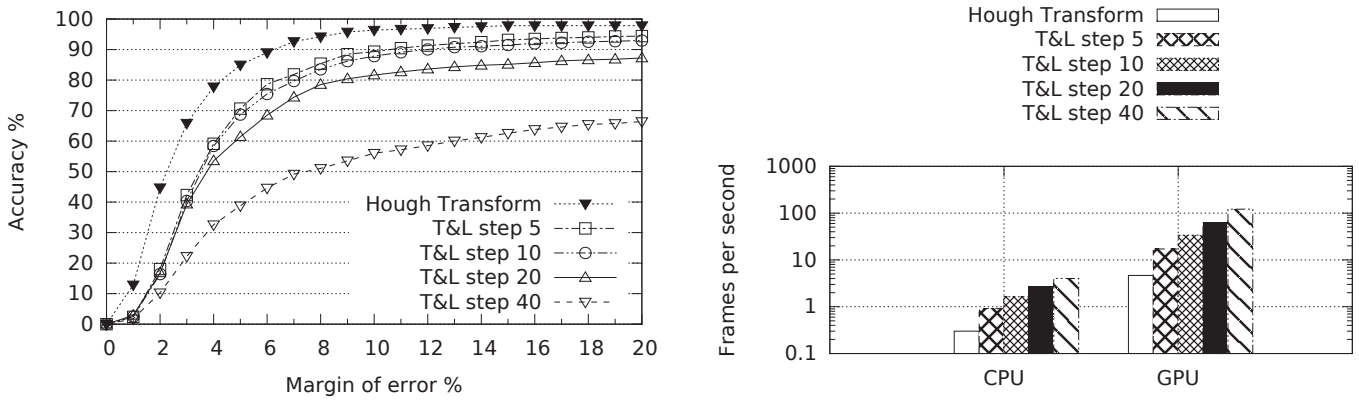
A comparison of the accuracy (Figure 6a) and the performance (Figure 6b) helps to understand how well the algorithms behave. For the sake of simplicity only the best configuration for both the Starburst and for Thresholding and Labelling is shown. For Starburst a corneal reflections removal kernel of size 19 and a Gaussian filter of size 5 and sigma 2 are used. For Thresholding and Labelling a step of size 20 is chosen.

When considering both metrics, performance and accuracy, it stands out that the best overall algorithm is Starburst. Its accuracy has been measured to be really high, with 96.2% of the pupils detected within a margin of error of 10%. Although the Hough Transform is just slightly better (96.38% of the pupils detected within a margin of error of 10%) it exhibits a significant lower performance, 4.7 frames per second, making



(a) (b)

Fig. 4: (a) Starburst's accuracy with different preprocessing configurations. (b) Starburst's performance for the same configurations.



(a) (b)

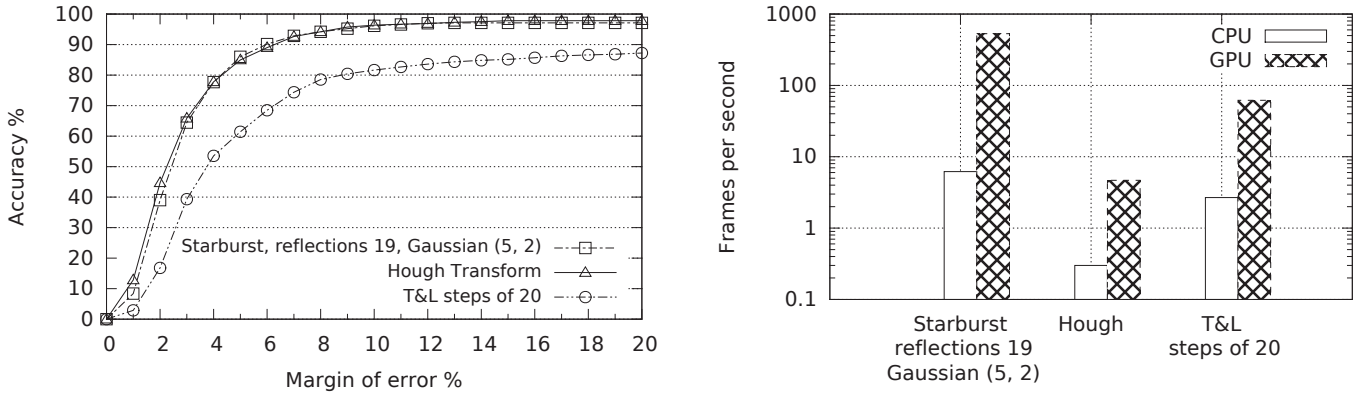
Fig. 5: (a) Hough and Thresholding and labelling accuracy with different step sizes. (b) Hough and Thresholding and labelling performance with different preprocessing configurations.

it unusable for real time applications. Contrarily, the Starburst algorithm achieves 530 frames per second for the mentioned accuracy. On the other hand, the Threshold and Labelling algorithm is much worse in terms of accuracy with 78.5% of the pupils detected within a margin of error of 10% (although other configurations of the T&L have shown a higher accuracy, again, at the cost of degrading too much the performance) and also worse in terms of performance when compared to the Starburst, achieving only 61.9 frames per second. Summarizing, the GPU-accelerated version of Starburst, after applying all the CUDA optimizations discussed in the paper, is by far the fastest approach (able to achieve more than 500 frames per second) while at the same time exhibiting a high accuracy.

VII. Conclusions

Looking at the previous side-by-side comparison it is clear that highly accurate and fast pupil tracking have been achieved which enables high speed tracking of the pupil with small errors. This has been possible with the Starburst algorithm that has been parallelized by using CUDA. The speedup obtained with CUDA is 85.5x which is much more than the 3.1x obtained with OpenMP. Indeed, the speed of the algorithm is so high that it is possible to perform high frequency tracking of fast movements of the eye, such as saccades, with high quality images. High-speed cameras capable of capturing hundreds of frames per second are widely available and they can be used in combination with our parallelized algorithm to do real-time tracking with high frame rates. Furthermore, in real-time environments usually some time must be spent on communication with the cameras and the actuators. So the GPU cannot be processing all the time, therefore, having an algorithm that is faster than what it is theoretically needed is a must in order to enable real-time processing when considering the overhead of the communication and synchronization.

The parallel implementation we have developed is a powerful tool that will help to improve visual optics applications which rely on pupil tracking, enabling speeds and a pupil tracking accuracy that were not possible previously. Furthermore, the stability requirements for the subject could be relaxed in some experiments.



(a) (b)

Fig. 6: (a) Comparison of accuracy of the algorithms. (b) Performance of the algorithms using a resolution of 1280x1024.

VIII. Acknowledgments

This research has been supported by the European Research Council Advanced Grant ERC-2013-AdG-339228 (SEECAT), by the Spanish SEIDI under grant FIS2013-41237-R, and by the Spanish MINECO under grant TIN2012-38341-C04-03.

References

[1] C. Hennessey, B. Noureddin, and P. Lawrence, "Fixation precision in high-speed noncontact eye-gaze tracking," *IEEE Trans. on Systems, Man, and Cybernetics, Part B: Cybernetics*, vol. 38, no. 2, pp. 289–298, 2008.

[2] C. Schwarz, P. M. Prieto, E. J. Fernández, and P. Artal, "Binocular adaptive optics vision analyzer with full control over the complex pupil functions," *OSA Optics Letters*, vol. 36, no. 24, pp. 4779–4781, 2011.

[3] D. Li, D. Winfield, and D. J. Parkhurst, "Starburst: A hybrid algorithm for video-based eye tracking combining feature-based and model-based approaches," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR) - Workshops*, 2005.

[4] R. Koprowski, M. Szmigiel, H. Kasprzak, Z. Wróbel, and S. Wilczyński, "Quantitative assessment of the impact of blood pulsation on images of the pupil in infrared light," *JOSA A*, vol. 32, no. 8, pp. 1446–1453, 2015.

[5] J. Canny, "A computational approach to edge detection," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[6] D. H. Ballard, "Generalizing the hough transform to detect arbitrary shapes," *Pattern Recognition*, vol. 13, no. 2, pp. 111–122, 1981.

[7] OpenMP Architecture Review Board, "OpenMP application program interface version 3.1," 2011.

[8] *CUDA C Programming guide*, Nvidia Corporation, 2015.

[9] A. A. Jarjes, K. Wang, and G. J. Mohammed, "GVF snake-based method for accurate pupil contour detection," *Information Technology Journal*, vol. 9, no. 8, pp. 1653–1658, 2010.

[10] H. Mehrabian and P. Hashemi-Tari, "Pupil boundary detection for iris recognition using graph cuts," in *Proc. of the Int'l Conf. on Image and Vision Computing New Zealand (IVCNZ)*, 2007, pp. 77–82.

[11] M. Soltany, S. T. Zadeh, and H.-R. Pourreza, "Fast and accurate pupil positioning algorithm using circular hough transform and gray projection," in *Proc. of the Int'l Conf. on Computer Communication and Management (CSIT)*, vol. 5, Sydney, Australia, 2011, pp. 556–561.

[12] L. Masek *et al.*, "Recognition of human iris patterns for biometric identification," Master's thesis, University of Western Australia, 2003.

[13] D. M. Rankin, B. W. Scotney, P. J. Morrow, D. R. McDowell, and B. K. Pierscionek, "Dynamic iris biometry: a technique for enhanced identification," *BMC research notes*, vol. 3, no. 1, p. 182, 2010.

[14] I. Borovikov, "Gpu-acceleration for surgical eye imaging," in *Proc. of the 4th SIAM Conference on Mathematics for Industry (MI09)*, San Francisco, CA, USA, 2009.

[15] J. B. Mulligan, "A GPU-accelerated software eye tracking system," in *Proc. of the ACM Symp. on Eye Tracking Research and Applications*, Santa Barbara, CA, USA, 2012, pp. 265–268.

[16] P. Majaranta and A. Bulling, "Eye tracking and eye-based human–computer interaction," in *Advances in Physiological Computing*. Springer, 2014, pp. 39–65.

[17] D. W. Hansen and P. Majaranta, "Basics of camera-based gaze tracking," in *Gaze Interaction and Applications of Eye Tracking: Advances in Assistive Technologies*. IGI Global, 2011, pp. 21–26.

[18] J. San Agustin, H. Skovsgaard, E. Mollenbach, M. Barret, M. Tall, D. W. Hansen, and J. P. Hansen, "Evaluation of a low-cost open-source gaze tracker," in *Proc. of the ACM Symposium on Eye-Tracking Research & Applications*, Austin, Texas, 2010, pp. 77–80.

[19] D. Li, J. Babcock, and D. J. Parkhurst, "openeyes: A low-cost head-mounted eye-tracking solution," in *Proc. of the ACM Symposium on Eye Tracking Research & Applications*, San Diego, California, 2006, pp. 95–100.

[20] C. Morimoto, D. Koons, A. Amir, and M. Flickner, "Pupil detection and tracking using multiple light sources," *Image and Vision Computing*, vol. 18, no. 4, pp. 331–335, 2000.

[21] W. Aydi, N. Masmoudi, and L. Kamoun, "New corneal reflection removal method used in iris recognition system," *World Academy of Science, Engineering and Technology*, vol. 5, no. 5, pp. 898–902, 2011.

[22] P. Chen, H. Zhao, C. Tao, and H. Sang, "Block-run-based connected component labelling algorithm for gpgpu using shared memory," *IET Electronics Letters*, vol. 47, no. 24, pp. 1309–1311, 2011.

[23] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[24] G. Taubin, "Estimation of planar curves, surfaces, and nonplanar space curves defined by implicit equations with applications to edge and range image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 11, pp. 1115–1138, 1991.

[25] S. Manzanera, P. M. Prieto, A. Benito, J. Tabernero, and P. Artal, "Location of achromatizing pupil position and first purkinje reflection in a normal population," *Investigative Ophthalmology & Visual Science*, vol. 56, no. 2, pp. 962–966, 2015.

[26] "cuRAND library, CUDA 7," *NVIDIA Corp., Santa Clara, CA*, 2010.

[27] V. Podlozhnyuk, "Image convolution with CUDA," *NVIDIA Corp. White Paper*, vol. 2097, no. 3, June, 2007.

[28] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.